Universitetet i Oslo
Institutt for informatikk

Monitoring Bluetooth network topology

Cand Scient Thesis

Fredrik Borg

February 1, 2002

II

Foreword

This thesis is a part of my cand.scient. degree in Communication Systems at the University of Oslo, Department of Informatics. The work has been done in collaboration with Telenor R&D, and my supervisors have been Dr. Do Van Thanh and Tore Erling Jønvik.

First of all, I would like to thank my supervisors for excellent guidance and interesting discussions. With their help I have also written two papers that were accepted at SCI2001, Orlando and CIC2001, Seoul. This was a great opportunity and a valuable experience.

During my work on the thesis I have also had great student fellows at Telenor. I have enjoyed both technical and non-technical discussion with Kjetil Marinius Sjulsen, Gunvald Martin Grødem, Erik Gjerdrum, Luis Flores and Anne Marie Hartvigsen. In addition to these persons I have had invaluable feedback and interesting discussions with Eivind Borg during the last stage of my thesis.

The last few days I had professional help from Lars Listerud and e-dit printing office to print the thesis. I want to thank them for fast and efficient support.

Last, but not least, I want to thank Anne Kjersti Røise for non-technical proofreading and moral support.

Fornebu, February 2002

Abstract

The Bluetooth technology is starting to be common in accessories like cellular phones and personal data assistants. Although Bluetooth was started as a project to replace the cables between cellular phones and its accessories, it is now seen as a more generic way of replacing cables between all devices. With Bluetooth, devices can exchange data without cables at distances up to 100 meters, and the user does not need to have the correct cable and plug to connect and exchange information between devices.

While the Bluetooth specification have advantageous characteristics like low power consumption and resistance to interference, the Bluetooth network topology can be difficult to follow. Bluetooth devices can be set up to initiate connections without user interaction, and devices can be connected to multiple devices at the same time. This makes it hard to know what the Bluetooth network topology looks like at a given time, and applications may not utilize the topology if there is no way to obtain information about it.

To solve this we introduce the concept of a Bluetooth network topology monitor. The monitor should be able to detect an initial network topology and changes that later occur in that topology. We first give a detailed description of Bluetooth and related technology so we can explore various methods the monitor can be constructed.

The work on this thesis has also resulted in two published papers:

- Borg, F., Thanh, D. V., Jønvik, T., "Monitoring Bluetooth network topology". Presented by Fredrik Borg and published in the proceedings at the 5th World Multi-conference on Systemics, Cybernetics and Informatics (SCI 2001), July 22-25 2001, Orlando, USA
- Borg, F., Thanh, D. V., Jønvik, T., "Monitoring Bluetooth network topology". Presented by Dr. Do Van Thanh and published in the proceedings at the 6th CDMA International Conference (CIC 2001), October 30. to November 2, Seoul, Korea

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we give some background information about this thesis and present our problem statement. The chapter also includes an overview of the different chapters.

1.1 Motivation

The possibility of connecting several devices together and hence extending the functionality has become an ultimate requirement from the users. However, using cable for connection is not convenient since correct cable and plug are required and the mobile user must have a large assortment of them when traveling. A cable itself imposes limitation on mobility and flexibility. The infrared link removes the burden of having a cable but it is not flexible since the connected devices must be placed in line with each other and the devices remain immobile.

With Bluetooth, devices can move within a range of 10 to 100 meters and stay connected. Although not yet widely deployed, the Bluetooth technology will presumably be a success, and lately there have been predictions about an explosion in the Bluetooth market. With the abundance of Bluetooth-enabled devices, it is reasonable to assume that Bluetooth will evolve from being a cable replacement to become a network infrastructure connecting multiple devices together.

1.2 Problem statement

The Bluetooth network infrastructure is of dynamic ad-hoc type, which is changing constantly and in arbitrary way depending on the movement of the Bluetooth-enabled devices. It would be desirable to get a more thorough understanding of the topology and its changes. Because a Bluetooth device

can be set up to automatically accept connections, it is also interesting to know whether there is a connection between two devices or not. If information about the Bluetooth network topology was available, it could also be utilized by networking applications (e.g. routing protocols). The topology information is required to implement the Internet Protocol (IP) over Bluetooth, and it is also useful to optimize the wireless ad-hoc network.

The problem statement that we will try to answer is:

Is it possible to construct a Bluetooth network topology monitor?

The terms used in the problem statement are explained here to prevent ambiguity:

- *Bluetooth network topology* - The Bluetooth devices and the connections between those devices
- *Bluetooth device* - A device with a Bluetooth antenna, Bluetooth stack and a unique 48-bit MAC address
- *monitor* - An application implemented in some type of device that detects an initial Bluetooth network topology and changes that later occur to that topology

We have divided our main problem statement into these sub-problems:

- What approaches can we use to construct a Bluetooth network topology monitor?
- What limitations do the different approaches have?

1.3 Methodology

The methodology we have used to answer the problem statement can be divided into two parts:

- Research of Bluetooth and related short-range wireless technologies
- Requirements specification, design and implementation of the Bluetooth network topology monitor

To answer our problem statement we had to study thoroughly the Bluetooth specification. The study was needed to get ideas of how a Bluetooth protocol could be implemented and discover the limitations in the Bluetooth specification. It was also valuable to study other related short-range wireless technologies to understand better the compromises that had been done in the Bluetooth specification.

After that we had studied the Bluetooth specification, we started to specify the requirements specification of the Bluetooth network topology monitor. While writing the requirements specification, we discovered two different approaches to the Bluetooth network topology monitor:

- hardware approach
- software approach

A design phase for both the hardware and software approach were carried out to identify the most appropriate solution. We implemented the first part of the solution we identified as most appropriate. This gave us valuable feedback to the theory obtained earlier in the thesis.

1.4 Outline of the thesis

Chapter 2 gives a description of the Bluetooth technology based on the Bluetooth specification. In this chapter we use the level of details required to understand the Bluetooth concepts used throughout the thesis.

Chapter 3 presents short-range wireless technologies that are related to Bluetooth in some way.

Chapter 4 is used to discuss the Bluetooth network topology monitor concept. In this chapter we also state the requirements for the monitor.

Chapter 5 discusses our hardware approach to the Bluetooth network topology monitor. Most parts of this chapter is used to present Bluetooth protocol analyzers and discuss whether ideas from these can be reused in the hardware approach to the Bluetooth network topology monitor.

Chapter 6 presents our second approach, the software approach. We discuss the limitations with such an approach and implements (appendix B) the first part of our solution.

Chapter 7 is the conclusion of this thesis. In addition to the conclusion it holds proposals for future work that could be valuable to investigate on the subject of Bluetooth network topology monitoring.

Chapter 2

The Bluetooth Technology

This chapter will give some background and introduction to the Bluetooth technology as described in the Bluetooth Specification [14]. The readers who are familiar with the technology can therefore skim this chapter. Please note that this thesis is written with the assumption that the reader has knowledge of Bluetooth as described in this chapter.

2.1 Why Bluetooth?

The aim of the Bluetooth technology has evolved from being an interface between mobile phones and their accessories to a generic cable replacement between all suitable devices.

Nowadays users have to deal with many types of cabling between devices. On a normal workstation PC there are at least cables to the mouse, keyboard and display and most people have other peripheral devices like printers, speakers, digital cameras and mobile phones. These cables add to the complexity of the system and, when moving devices around, also restrict mobility. If devices could exchange information without cables it would also extend the possible applications for these devices, such as automatic synchronization when devices come into range. This shows that for the user there are many advantages with wireless communication.

Today there are many specifications of wireless technologies and devices based on these specifications are starting to hit the market. You can find a discussion about other wireless technologies in chapter 3, but here we will summarize some of the aspects that might help Bluetooth survive in this market:

- **Supported by major companies:** The Bluetooth Bluetooth Special Interest Group (SIG) has about 2500 members including all major wireless actors in the market.

- **Open Specification:** The Bluetooth specification [14] is an open document free for everyone to read.
- **Low power:** Bluetooth has several features that result in low power usage (see section 2.5.2). This way it can be included in small devices such as mobile phones and personal data assistants.

2.2 History

The Bluetooth technology was born in 1994 when Ericsson Mobile Communications initiated a study to investigate the feasibility of a low-power, low-cost radio interface between mobile phones and their accessories. The Bluetooth name comes from the tenth-century Danish Viking Harald Blåtand who united Norway and Denmark and it was chosen because Bluetooth wireless technology is expected to unify the telecommunication and computing industry. In February 1998 Ericsson formed the Bluetooth SIG together with Nokia, IBM, Toshiba and Intel. The Bluetooth SIG is an organization open for all companies committed to support the Bluetooth Specification.

The first version of the Bluetooth Specification (V1.0) was released July 1999 and was followed by V1.0B in December 1999 and V1.1 February 2001.

2.3 Where is Bluetooth today?

In 2000 Cahner In-Stat Group predicted that there would be 1.4 billion Bluetooth enabled devices in use within five years. One year later this number had been adjusted to 955 million units in 2005.

As of January 2002, Bluetooth chips are integrated in some high-end cellular phones and can be bought as an add-on to many personal data assistants (PDAs).

2.4 The Bluetooth Stack

A schematic figure of the Bluetooth stack is shown in figure 2.1. In the following sections we present the various layers in the stack, starting at the bottom.

2.5 Radio

Bluetooth devices operate on the globally available 2.4GHz Industrial Scientific Medical (ISM) band. The usage of the ISM band is unlicensed in most countries and Bluetooth must therefore cope with interference from:

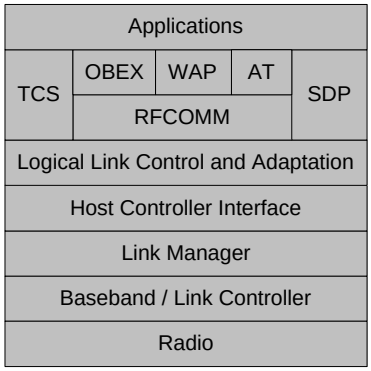

Figure 2.1: The Bluetooth Stack

- Other radio frequency short-range proprietary techniques (i.e. car security and cord-less phones) that uses the 2.4GHz spectrum
- Wireless local area networks (e.g. IEEE 802.11)
- Random noise generators (e.g. microwave ovens)
- Other Bluetooth units

A supplementary discussion regarding interference can be found in section 3.1.3 where we discuss coexistence between short-range wireless technologies.

In order to transmit and receive in the possibly noisy ISM-band Bluetooth employs several techniques to minimize packet loss: frequency hopping, adaptive power control and short data packets. Frequency hopping and power control are discussed in the following sections, while the Bluetooth data packet structure is explained in section 2.6.4.

2.5.1 Frequency Hopping

Bluetooth uses a Frequency Hopping Spread Spectrum (FHSS) scheme which divides the ISM-band into 79 1-Mhz channels[1]. The communication between devices switches between the available channels so interference with other Bluetooth and non-Bluetooth devices are kept to a minimum (figure 2.2).

In a normal Bluetooth connection all the 79 channels are used, but a 32 frequency hopping scheme (spanning the 79) are used in inquiry and paging (section 2.7).

Although Bluetooth provides retransmitting of lost packets, the source of interference will most likely also be present at the same frequency on the

[1]In France and Spain the ISM band is not fully unlicensed and only 23 channels are available. Some functions differs in the 79 and 23 frequency hopping schemes, but in this thesis we will only focus on the 79 channel version of these functions.

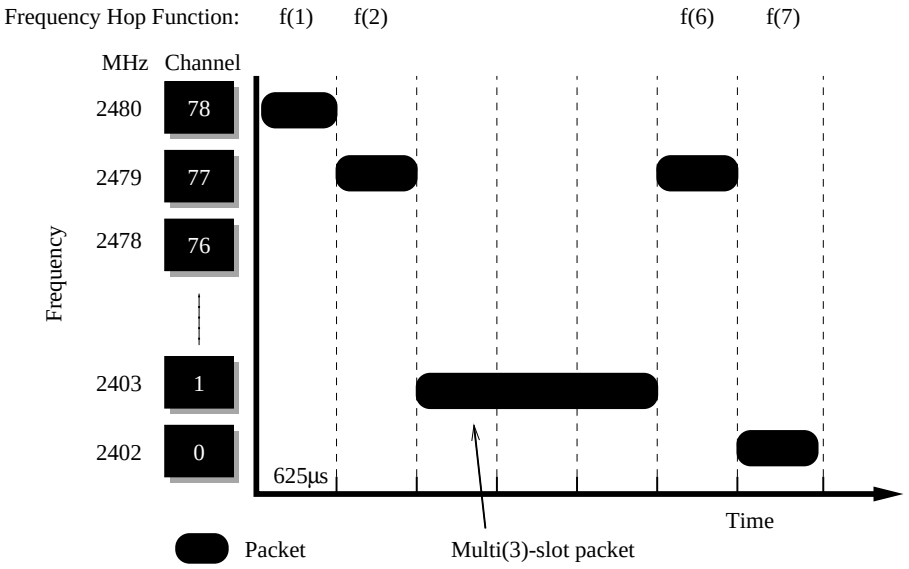

Figure 2.2: Frequency Hopping Spread Spectrum

next time-slot. Transmitting of packets on a different frequency will therefore have a higher chance of succeeding. Indeed, the algorithm used to calculate the hop sequence ensures maximum distance between adjacent hop channels [6].

2.5.2 Power Control

In the Bluetooth specification there are defined three power classes with max output power from 1mW (Class 3) to 100mW (Class 1) and effective communication range from 10 to 100 meters. Bluetooth devices may also implement power control to optimize power consumption and decrease interference. With power control two devices communicating measures the Received Signal Strength Indication (RSSI) and asks the other device to decrease/increase the output power based on this.

2.6 Baseband

The baseband is responsible for channel coding/decoding, timing and managing a Bluetooth link for the duration of one packet.

2.6.1 Master/Slave

Bluetooth devices in a connection are either master or slave. In a single point-to-point connection one device is master and the other device is slave.

Point-to-multipoint connections are also allowed and here one device will be master and all the other devices slaves.

2.6.2 Piconet/Scatternet

On the baseband level, communication is only possible between a master and its slaves. If two slaves want to speak directly to each other they have to make their own piconet. A master and the slaves it controls are named piconets (figure 2.3a). Multiple piconets connected together, either by slaves that are members of more than one piconet (figure 2.3b) or by a slave in one piconet that is master in another piconet (figure 2.3c), are named scatternets.

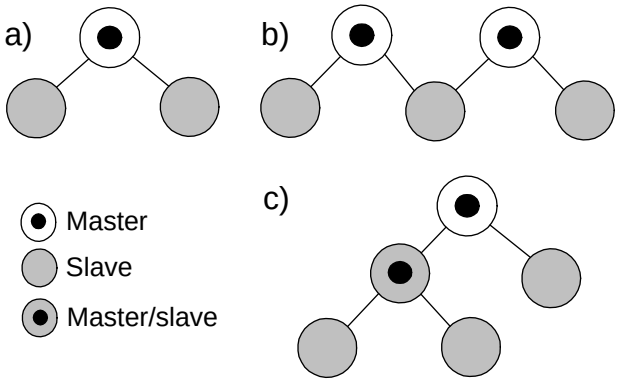

Figure 2.3: Piconet (a) and Scatternet (b and c)

2.6.3 Addressing

All Bluetooth Devices have a unique 48-bit IEEE MAC address, the Bluetooth Device Address (BD_ADDR). In addition there are unique addresses within one piconet, the 3-bit Active Member Address (AM_ADDR). The AM_ADDR is assigned by the master to each of the slaves connected to it, and 3-bit gives the master the possibility to control up to 7 slaves (000 is the broadcast address).

2.6.4 Bluetooth Packet Structure

Figure 2.4 shows the generic format of all Bluetooth packets.

In this section we will briefly explain the structure of the Access Code and the Header, which is of most importance in this thesis. We will also present the special Frequency Hopping Sequence (FHS) packets which are used during device discovery and connection establishment.

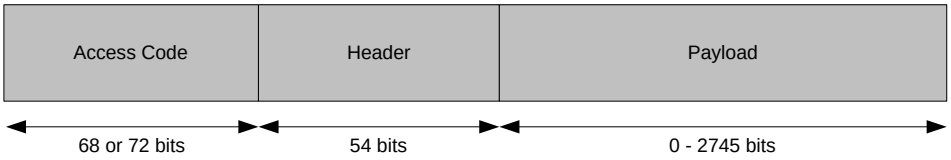

Figure 2.4: Bluetooth Packet Structure

Access Code

The Access Code is used for synchronization and identification. A Bluetooth receiver listens on the air interface for the *Preamble* (figure 2.5) and decodes the *Synchronization Word* that follows.

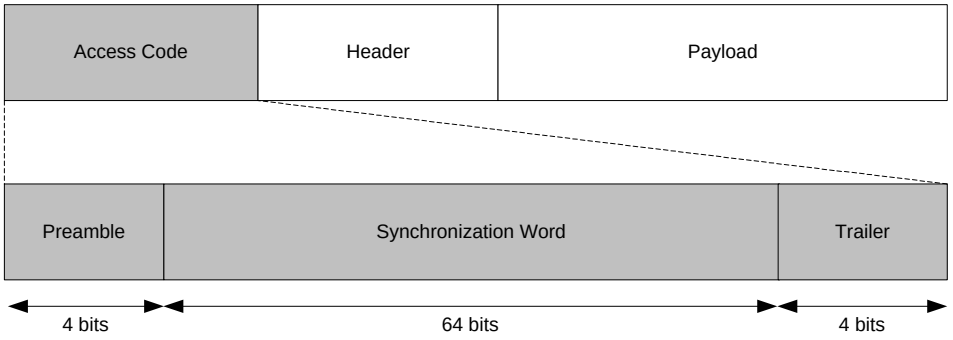

Figure 2.5: Bluetooth Access Code Structure

The Synchronization Word is the same for all packets sent within a piconet and is constructed based upon the BD_ADDR of the piconet master. If the master is the receiver of a packet, it will know that from the synchronization word. However, if the destination is a slave, all slaves in the piconet must continue to read the header to decode the AM_ADDR.

The Access Code also includes 4 trailer bits used when synchronizing but this is only included if a payload is included in the packet. This explains why the Access Code can be either 68 or 72 bits.

Header

The Bluetooth packet header is shown in figure 2.6. We will not cover the complete details of the header here, please refer to the Bluetooth specification [14]. The important thing for us is that the header contains the AM_ADDR which shows the AM_ADDR of the slave, whether it is a slave-to-master or master-to-slave packet.

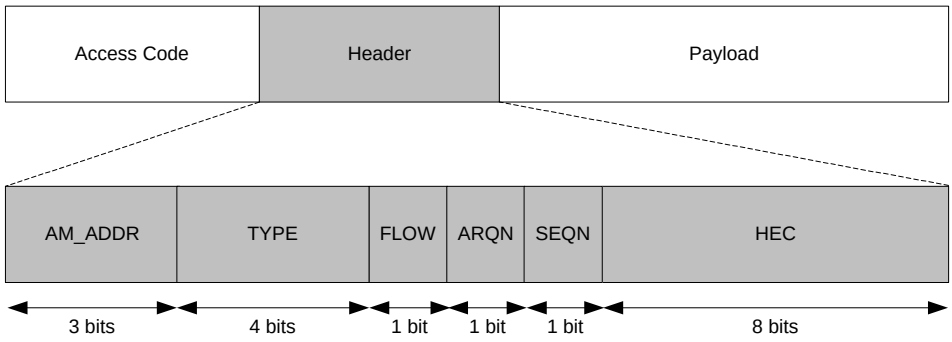

Figure 2.6: Bluetooth Header Structure

The FHS packet

The Frequency Hopping Sequence (FHS) packet is a special packet that is used to exchange synchronization information between Bluetooth devices. The packet is illustrated in figure 2.7.

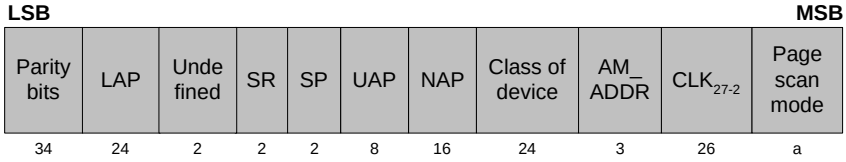

Figure 2.7: FHS Payload

In table 2.1 we give the description of the FHS fields that are relevant to us. For a complete description of the various fields, see the full specification [14].

Field	**Description**
Lower Address Part (LAP), Upper Address Part (UAP) and Non-significant Address Part (NAP)	Together these fields forms the BD_ADDR of the device that sends the FHS packet
AM_ADDR	In a connection setup or a master/slave switch this field contains the AM_ADDR that the recipient gets
CLK_{27-2}	The native clock of the device that sends the FHS packet

Table 2.1: FHS field descriptions

2.6.5 Bluetooth Clock

All Bluetooth devices have a free running clock, the *native clock*, with resolution of at least 312.5μs (half a time slot). This clock is never adjusted and will wrap about once a day. To synchronize to other Bluetooth units offsets are added to the native clock which provides temporary mutually synchronized clocks.

2.6.6 Physical Links

Bluetooth defines two types of physical links, both with various types of error protection.

Synchronous Connection Oriented (SCO)

SCO links use slots that are reserved by the master and are therefore considered to be circuit switched. Data sent over a SCO link are never retransmitted, because SCO links are used by audio where retransmission is of no use.

Asynchronous Connection-Less (ACL)

In the slots not reserved to SCO packets the master may freely exchange ACL packets with its slaves. Because of this, ACL links are considered to be packet switched. ACL links are used to transfer user data and control data and usually includes a Cyclic Redundancy Check (CRC). If the CRC fails at the destination, the packet will be retransmitted.

2.7 The Link Manager and Link Controller States

The *link manager* is responsible for establishing, supervising and tear down connections and logical links. To carry out these tasks the concept of *link controller states* are introduced.

There are two main states a Bluetooth device can be in; *standby* and *connection*. In addition there are several sub-states used when establishing a piconet (figure 2.8). In this section we will thoroughly explain the various states and the different transitions between them. This section provides essential background information to chapter 5 where we discuss Bluetooth synchronization.

2.7.1 Link Controller State Diagram

The Link Controller States defines how Bluetooth devices can discover other Bluetooth devices and how they connect to the discovered devices. In figure 2.8 we show the different states and the transitions between them. Please

note that not all indirect state transitions in this figure are possible. E.g. it is not possible to go from the standby state to the connection state via *inquiry* (it had to go via *page*). The diagram is drawn in this way to show that it is possible to go to inquiry from both the standby and the connection state.

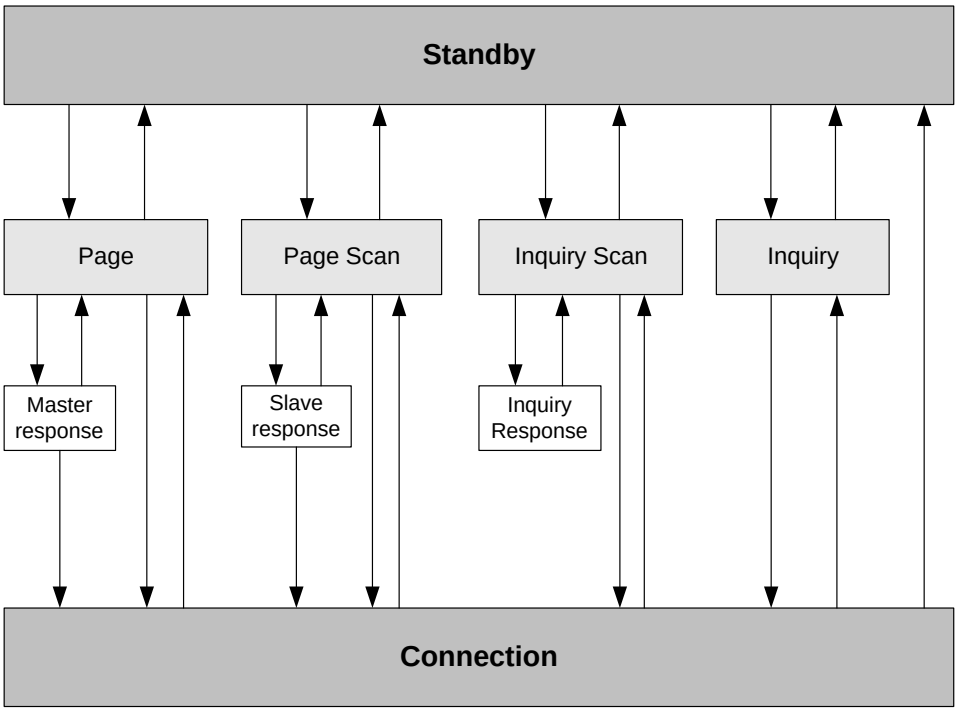

Figure 2.8: State Diagram

2.7.2 Standby

The Standby state is the default state in all Bluetooth units, and it is the state where the device uses least power. There are no connections to other Bluetooth devices in this state, and the only job performed is the update of the native clock.

2.7.3 Inquiry / Inquiry Scan

Inquiry is also called device discovery, because it is often used to detect all devices in an unknown environment.

Figure 2.9 shows that the device discovering nearby devices performs the *inquiry*, while the device(s) that wants to be discovered must be in the *inquiry scan* state. In this figure, device A will discover device B and C, but device D is not in inquiry scan mode and will not be discovered by device A.

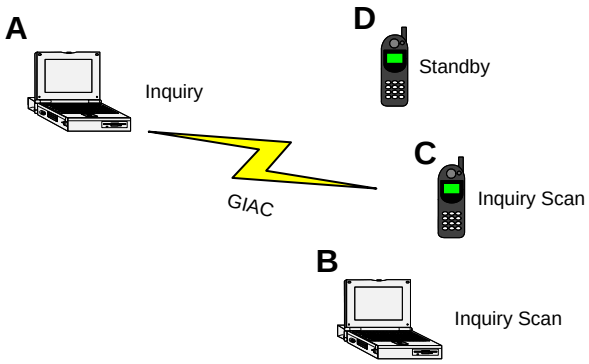

Figure 2.9: Device Discovery with Inquiry / Inquiry Scan

In the inquiry process the inquirer (the device performing the inquiry) will send out ID packets containing an Inquiry Access Code (IAC). The IAC is usually the General Inquiry Access Code (GIAC) and although other IACs are defined the GIAC is the only one that is relevant for us. An inquirer will in the inquiry process use a predefined frequency hopping scheme with 32 hops. In addition it will hop twice as fast as normal, thus transmitting the GIAC every 312.5μs. An example of a subset of an inquiry is shown in figure 2.10. In this figure the inquiry reach the inquiry scanning device the second time and responds to the inquiry after the random back-off period (explained in the next section).

While the inquiry will hop twice as fast as normal, the device in inquiry scan will only hop once each 1.28s, but using the same predefined frequency hopping pattern as the inquirer. When a device receives the ID packet in an inquiry it moves to the *inquiry response* sub-state. In this sub-state the device will respond to the inquiry. It is important that this response is not sent immediately, because this could cause unnecessary interference. Instead it waits a random back-off period (0-2047 time-slots) before a response is sent. The response is a normal FHS packet and it is received by the inquirer in between the GIAC transmissions (figure 2.10). Note that the inquiry response is not acknowledged by the inquirer.

As stated, the goal of the inquiry is to discover nearby device, but more precise the inquirer will collect the following information from the respondents:

- BD_ADDR (the unique 48bit IEEE mac address)
- Native clock

Information about the remote native clock is optional in a connection attempt but knowledge of it will speed up the connection process (more details

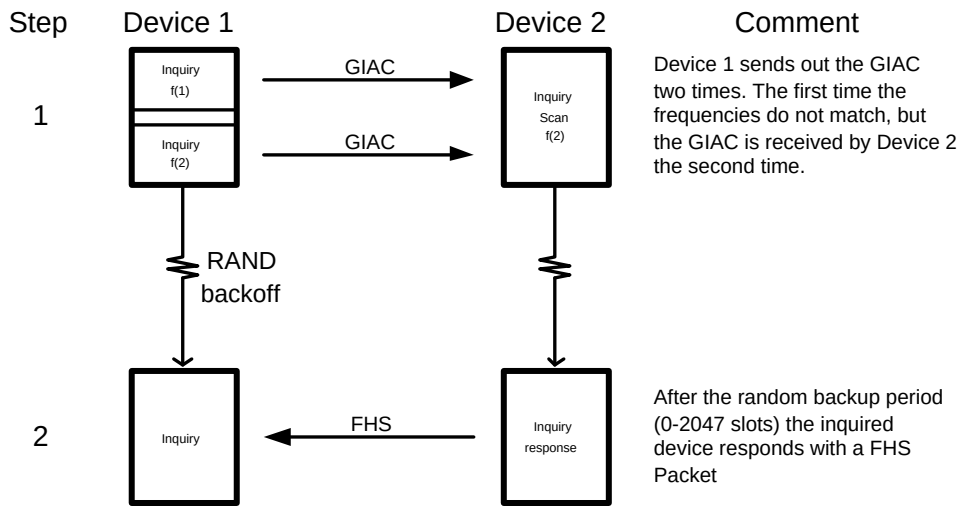

Figure 2.10: Inquiry and response

in section 2.7.4). The BD_ADDR is however essential information in a later connection attempt.

Inquiry in an error-free environment where the inquirer has no active SCO links usually lasts 10.24s and the details and calculation are given here:

- The GIAC will be sent out in 10ms *trains* spanning a predefined 32 frequency hopping scheme.
- The trains should be repeated 256 times before a new train is used and at least three train switches must take place to make sure all responses (FHS packets) are collected.
- $10ms * 256 * 4$ (three switches) $= 10.24s$

If the inquiry is carried out in an error-prone environment or with active SCO links, more time must be used by inquiry to make sure all devices are discovered. Bray [6] have calculations which show that 60% and 90% of the scanning time is lost with respectively one and two active SCO links.

An inquiry will occupy most of the bandwidth in the inquiry device while the inquiry is carried out. ACL links (section 2.6.6) should be put on hold, but SCO links (section 2.6.6) uses reserved bandwidth and should not be touched.

2.7.4 Page / Page Scan

Paging describes how Bluetooth devices establish connections. Prior to the paging the paging device, the device initiating the connection, must obtain the BD_ADDR of the device it intend to make a connection to. This is usually achieved through inquiry, but it can also be factory preset (e.g. an earphone that is produced to work with a fixed cellular phone) or entered by a user.

Device A in figure 2.11, that are in the Page State, transmits the Device Access Code (DAC) of device C. The page process can be compared to the inquiry process, except that in paging we are only interested in answers from one device, the device we want to establish a connection to. Only device C will respond to the page request in the scenario presented in figure 2.11.

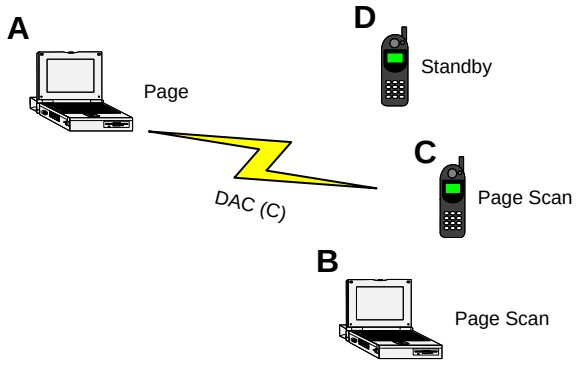

Figure 2.11: Connection establishment with Page / Page Scan

To let a paging device reach a page scanning device they must at some point transmit at the same **frequency** and **time**.
In the inquiry case the frequency hopping used a predefined pattern known by all Bluetooth devices, but this is not done in paging. Instead a pattern is constructed from a function that takes the BD_ADDR of the page scanning device as input parameter:

$$g(\text{BD\_ADDR}) = \text{32-hop sequence}$$

This shows why the paging device has to know the BD_ADDR of the remote device. Without the BD_ADDR it does not know how the remote device is hopping. The function is rather complex and will not be discussed here. What is important to us is the result of the function, a 32-hop sequence spanning the 79 Bluetooth hop frequencies.

We have now shown how the device is synchronized in frequency, but synchronization in time is also necessary. This process will be executed faster if we have a good estimate of the page scanning device's native clock obtained from e.g. a recent inquiry. The estimate is named $f(k)$ and two trains of frequencies are derived (table 2.2).

Train A $= f(k-8), f(k-7), ..., f(k+7)$
Train B $= f(k-16), f(k-15), f(k-9), ..., f(k+8), ..., f(k+15)$

Table 2.2: Page trains

Train A consists of the 16 frequencies surrounding the clock estimate, $f(k)$, and train B consists of the 8 frequencies before and after train A (figure 2.12).

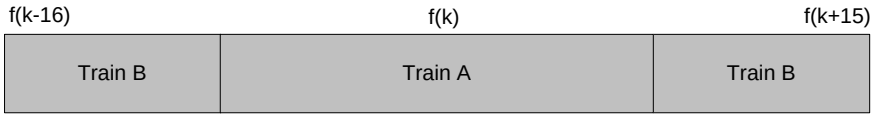

Figure 2.12: Train A and Train B

To explain the page and page scanning we have constructed an example that is shown in figure 2.13. In this figure the state of the device is included in the boxes as well as the hop sequence in the page and page scan states. We will explain the various steps in that figure and compare the process with the inquiry process where that is appropriate.

To achieve as fast connection establishment as possible, the paging device transmits on the frequencies on Train A first. If the clock estimate is good, this will be successful. In our example (step 1), the paged device receives the page request already on the second attempt. The request includes the destinations DAC and this is also returned as an acknowledgement to the paging device (step 2). In the next step the paging device transmits a FHS packet (step 3) that lets the paged device synchronize to the paging device. When the FHS packet is acknowledged (step 4), the synchronization takes place. After this synchronization, the connection is established and a poll packet is sent (step 7) and acknowledged (step 8) to verify that the synchronization was successful.

Figure 2.13 shows an almost optimal, but not necessary common page procedure. The paging spans a couple of time slots and this may be the case if the clock estimate is good. If the clock estimate is bad or not available, the page procedure will need more time. The A and B trains can be retransmitted

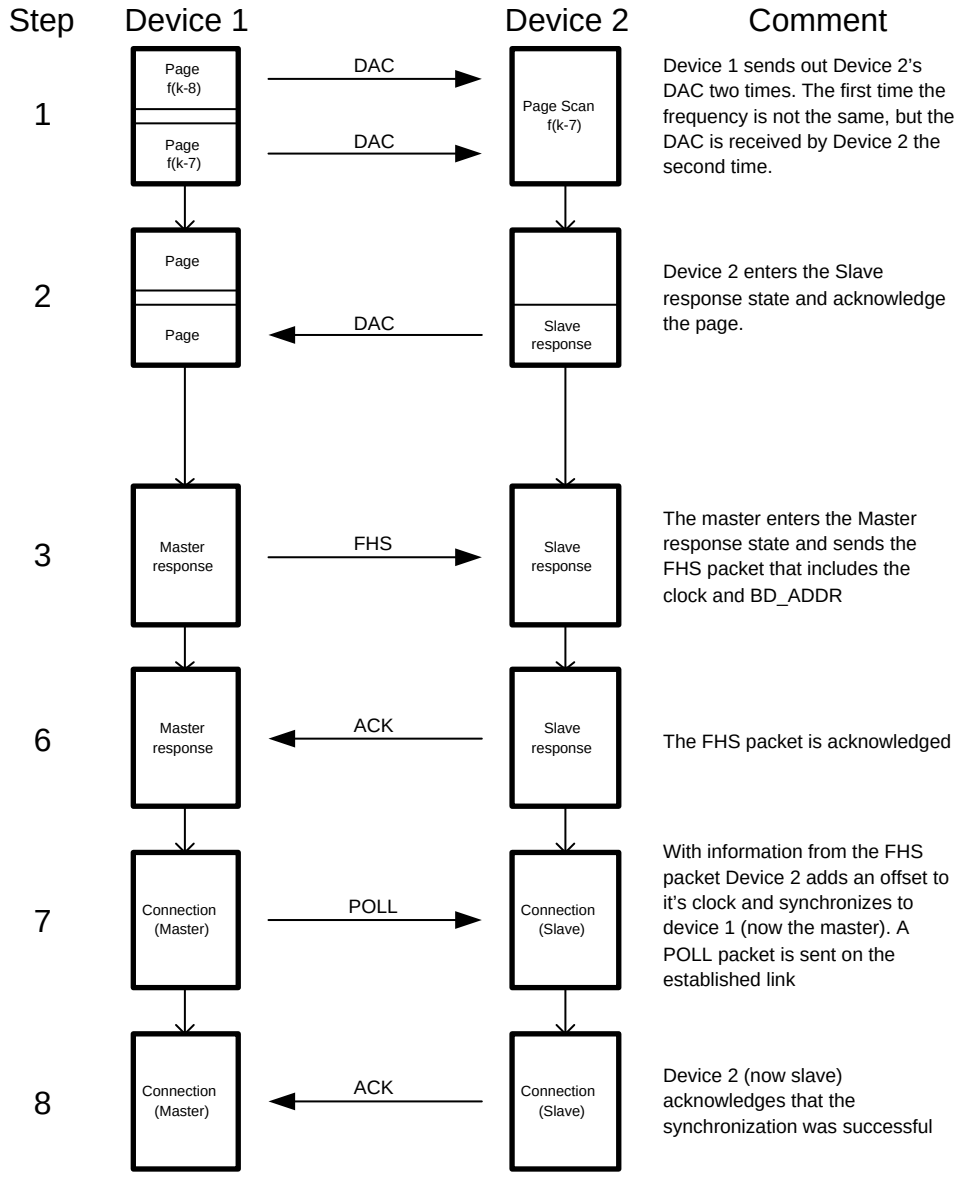

Figure 2.13: Example of Page / Page scan

up to 128 times before a connection is accomplished. In the page state the hopping is twice as fast as normal hopping, but a time slot is used to listen for page response after every two page-packets that are sent. Worst case in page scanning is thus $625\mu s * 32 * 128 = 2.56s$.

2.7.5 Connection

In the connection state the master and slave are synchronized and the connection established. The connecting device is now the master and the other device the slave. In the connection state the slave is synchronized to the master and thus follows the frequency hopping of the master.

2.8 Host Controller Interface (HCI)

The HCI is provided to ease the partition of the Bluetooth Stack across two processors. Some systems (such as a PCMCIA card on a laptop) will implement the baseband and link manager on the Bluetooth device and higher levels on the host processor. The HCI is provided as an interface between these parts and can help drivers to be written for Bluetooth hardware from different manufacturers. A schematic figure of the HCI from Bray [6] is given in figure 2.14.

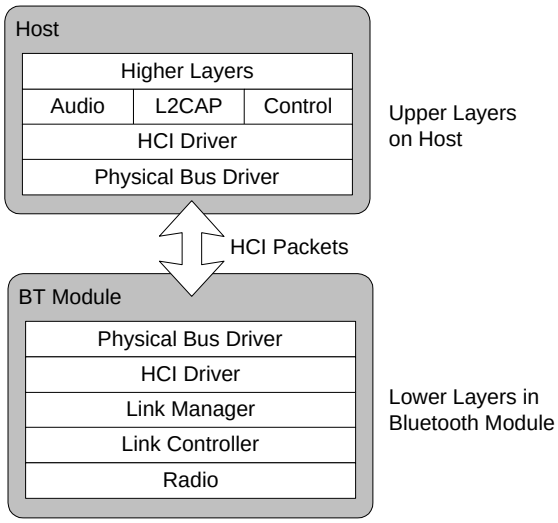

Figure 2.14: The Host Controller Interface

2.9 Logical Link Control and Adaption (L2CAP)

L2CAP deals with multiplexing of different services, segmentation and reassembling of packets and Quality of Service (QoS). It is positioned above HCI as shown in figure 2.14, and ACL links are used to carry out the transportation.

Multiplexing in L2CAP helps higher level applications share a common ACL-connection. The segmentation and reassembling part of L2CAP hides the Max Transfer Unit (MTU) limits of lower layers and makes it possible to transfer packets up to 65535 bytes. QoS in L2CAP gives higher layer applications the possibility to reserve bandwidth and fulfill latency and jitter (latency variation) demands.

Details of the L2CAP will not be given here, but can be found in the Bluetooth specification [14] and in Bray [6].

2.10 Profiles

The Bluetooth Profiles [15] provide interoperability between devices from different manufacturers for specific services and use cases. A profile defines a selection of messages and procedures (generally termed capabilities) and gives an unambiguous description of the communication between two devices. Two communicating devices shall use the same profile. At this time there are 2 profiles that most devices share, the Generic Access Profiles and the Service Discovery Profile. In addition to these profiles there are 11 profiles that deal with different usage scenarios. SIG will later publish more profiles which will further extend the range of applications.

2.10.1 Relationships between profiles and services

The profiles defined by the Bluetooth SIG are only descriptions of functionality provided by a Bluetooth device. In other words, a profile is a specification of how a certain service must be implemented. A service, according to the Bluetooth Software Suite SDK [10], is an instance of the implementation of a profile.

2.10.2 Generic Access Profile (GAP)

The Generic Access Profiles defines the general procedures needed to discover and make connections to other Bluetooth enabled devices. All other profiles depend on the GAP to do these basic functions and all Bluetooth devices must at least support this profile. The relationships between different profiles are further shown in figure 2.15. If square X is within square Y, it means that profile X depends on profile Y.

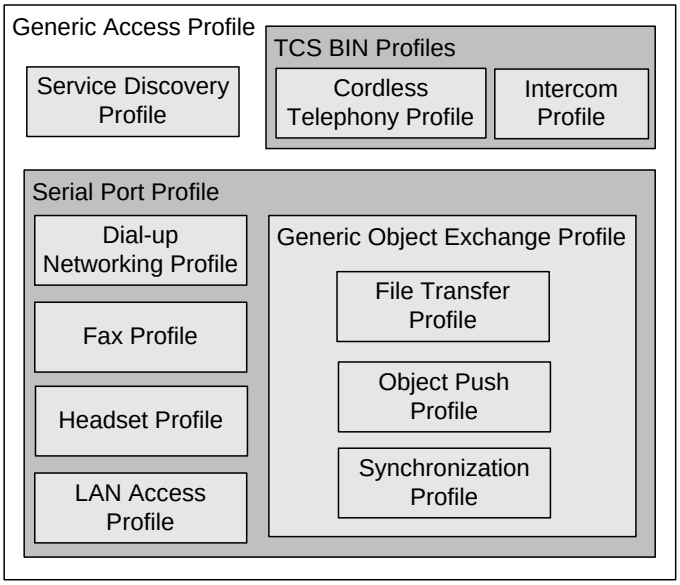

Figure 2.15: Relationship among different profiles

As we see in figure 2.15, all profiles depend on the GAP, and there are many profiles that are built on top of the Serial Port Profile.

2.10.3 Service Discovery Profile (SDP)

When a device has performed an inquiry and discovered nearby discovery devices, it usually does a Service Discovery to get knowledge of the service(s) the discovered devices provides. The SDP gives devices the opportunity to discover registered services in other devices and retrieve information about these services. The SDP specification does not define methods to access the services, only discover them. Accessing them should be implemented in other ways.

SDP Basic methods

With the SDP a server application may define and notify the availability of a new service. When using SDP, a server application may:

- Define and notify about the availability of a new service

A client application will get the opportunity to:

- Search for a given service (specified by attributes)
- Browse available services.

SDP Requests

The client application can issue an SDP request via its SDP Client. An SDP Server associated with a Server Application will pick up the request and try to match the request with the services it provides and give a response. This is illustrated in figure 2.16.

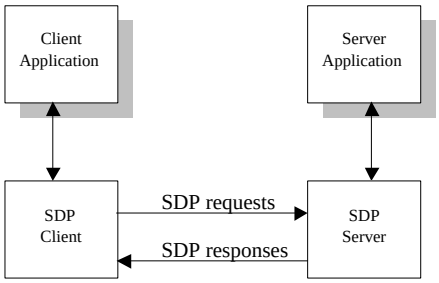

Figure 2.16: SDP Requests

2.10.4 Other profiles

No detailed description of other profiles will be given here, but a short summary of the profiles included in the Bluetooth 1.1 specification [15] is given in table 2.3.

In addition to the published profiles shown in table 2.3 there are numerous profiles that are currently in the draft version. These profiles are not yet released to the public, but are available for early adapters and associate members of the SIG. Some of these profiles are listed in table 2.4 to give the reader some information about the current work in the Bluetooth development. Note that although these profiles are not yet released to the public, some of them are already implemented in consumer products.

2.10.5 Master/Slave roles in different profiles

When a device makes a connection to another device, the device initiating the connection will always be the master and the device accepting the connection will be slave. There are however scenarios where the initial roles are not optimal. This is normally the case with devices providing a service to multiple devices at the same time, i.e. acting as a gateway.

A gateway will normally not know which devices that wants to use its services, and it is the clients that have to initiate the connection to the gateway. All clients connecting to the gateway will initially be masters, and the gateway will be slave in several piconets, one piconet for each of the clients using

Profile	Description
Cordless Telephony Profile	Cordless telephony using a cellular phone and an access point or as a replacement for DECT and other systems.
Intercom Profile	E.g. direct connection between Bluetooth enabled cellular phones.
Serial Port Profile	Serial cable emulation.
Headset Profile	Headset to connect to e.g. cellular phone
Dial-Up Networking Profile	Bluetooth as an Internet-bridge, e.g. using a laptop to connect to the Internet via a Bluetooth enabled cellular phone.
Fax Profile	Same as Dial-Up Networking, but for FAX services.
LAN Access Profile	LAN Access for one or more Bluetooth devices.
Generic Object Exchange Profile	Defines a generic way of exchanging objects over a Bluetooth link.
Object Push Profile	Defines the process to push and pull objects (e.g. exchange business cards).
File Transfer Profile	Provides methods to browse, transfer and manipulate files on a Bluetooth connection.
Synchronization Profile	Synchronization of e.g. calendar and address book.

Table 2.3: Profiles

Working Group	Description
Personal Area Networking	Ad-hoc IP-based networking between Bluetooth enabled devices.
Printing	Bluetooth enabled printers.
Human Interface Device	Mouse-, keyboard- and game controller Bluetooth enabled devices.
Richer Audio/Voice/ Video	Support for hi-fi audio/video/voice over Bluetooth

Table 2.4: Draft Profiles

its services. It would be feasible to use the point-to-multipoint functions from Bluetooth, i.e. let the gateway be master and the clients slaves.

In some Bluetooth profiles (Cordless Telephony Profile and LAN Access Profile) this is achieved by having the Gateway accept connections from devices and time-share by being both master and slave, but try to switch role with the connecting device as fast as possible (figure 2.17). The connecting device will be disconnected if it refuses a master/slave switch.

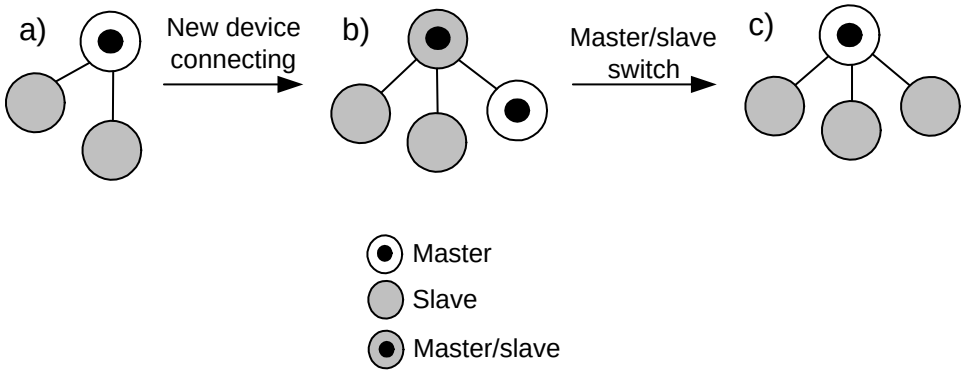

Figure 2.17: Master/slave switch with gateways

If a device should act this way, the master/slave roles must be defined in the profile. In the Bluetooth 1.1 Specification [15] the master/slave roles are only specified in the Cordless Telephony Profile (CDP) and the LAN Access Profile. In all the other profiles the roles are irrelevant, because they only support point-to-point communication.

2.11 Comments to the Bluetooth specification

While studying the Bluetooth specification we have identified some issues that limit the usage of Bluetooth in some way. In this section we briefly comment these issues.

Speed

In version 1.1 of the Bluetooth specification the max, raw transfer rate is 1 Mb/s, which is considerable slower than most other short-range wireless technologies. There is a working group that deals with this issue and the next version of the Bluetooth specification is believed to include an optional, backwards-compatible high-speed extension.

Connection-time

A Bluetooth device will spend approximately 10 seconds to detect available devices in an unknown environment and 1-2 seconds to connect to a known device. In some cases this may be to long, e.g. if the devices are moving at a high speed.

Scatternets

Scatternets are included in the current specification as a way of linking several piconets together. At this time there are however few Bluetooth stacks that supports scatternets.

Profiles

Bluetooth relies on the concept of profiles to assure that devices produced by different manufactures shall be able to talk to each other. There are two pitfalls arising regarding interoperability:

- Full interoperability between standard SIG Profiles
- Adoption of new (proprietary) protocols

If devices from different manufacturers should be able to talk to each other, it is important that designers of the devices interpret the specification in the same way. This does not only apply to the profiles, but also to the rest of the Bluetooth specification. One of the reasons the Bluetooth Specification V1.1 was released was in fact that some issues regarding authentication was not specified strictly enough in V1.0B. Different manufacturers had interpreted the authentication steps differently and devices from such manufacturers were not able to establish connections to each other.

As new user scenarios are introduced, there will probably be a need to introduce new profiles. However, it is an open issue how new profiles should be standardized. The Bluetooth SIG is today handling the development of new profiles, but time will show if they manage to develop new profiles quickly enough to let the Bluetooth technology evolve.

Ad-hoc Bluetooth networks

Bluetooth devices are believed to connect to other devices in an ad-hoc fashion and it can be hard to know what connections are active at a given time. If a scatternet is created, it is also impossible to get an understanding of the scatternet topology, because device will only have knowledge of their local connections.

This issue is in fact the topic of this thesis and is discussed in greater detail throughout chapter 4, 5 and 6.

Solutions

The solution to some of the issues raised here is simply to use another technology. What we mean by that is that Bluetooth is not the ideal approach for every scenario. When the Bluetooth specification was developed, many compromises had to be done. One example is the compromise between speed and resistance to interference. With the frequency hopping scheme used by Bluetooth less bandwidth is available for each device, but several devices at the same location will less likely experience interference.

For a presentation of other available short-range technologies to consider, please refer to chapter 3.

Some of the issues discussed here will also be addressed in later revisions of the Bluetooth specification and are already the subject of different Working Groups in the Bluetooth SIG.

Chapter 3

Short-range Wireless Technologies

In this chapter we give a description of technologies that can potentially be competitors to Bluetooth. The technologies that we present are:

- IrDA (infrared)
- Wireless LAN (IEEE 802.11)
- HIPERLAN2
- HomeRF
- SPIKE

Regarding these technologies we describe differences and similarities with the Bluetooth technology where it is appropriate.

3.1 Characteristics of wireless communication

Before we go into details of the various technologies we summarize some characteristics of short range wireless technologies.

At this time there exists numerous specifications of different wireless technologies, and it is hard to say which are here to stay and which will be replaced by other technologies. Some researchers have argued that Bluetooth do not have any real competitors except cables [26] while some journalists argues that Bluetooth already have lost the market to IEEE 802.11 [16]. We will not continue that discussion here, but instead present wireless technologies that share some characteristics with Bluetooth, or share some range of applications.

3.1.1 Frequencies

The common dividend for all technologies described in this chapter is that they do not use cables to exchange information. Packets are modulated to be sent as electromagnetic waves on the air and are demodulated at the other end.

The sender and receiver have to agree on what frequency to transmit the electromagnetic waves. Short range wireless systems can use the infrared portion of the electromagnetic spectrum or one of the ISM frequencies (902-928 MHz, 2400-2483.5 MHz and 5725-5850 MHz). The ISM frequencies are free to use in most countries, but the disadvantage is that devices operating in this spectrum must cope with interference from other devices (section 3.1.3).

Because interference can be a big problem, some frequency spectrums are reserved for different purposes. One example is the 5150-5350 MHz that in many countries, including Norway, is reserved for indoor wireless networks, e.g. HIPERLAN.

3.1.2 Peer-to-peer and Access Points

Communication between wireless devices are either done directly (peer-to-peer, see figure 3.1) or through an access point (client/server model, see figure 3.2).

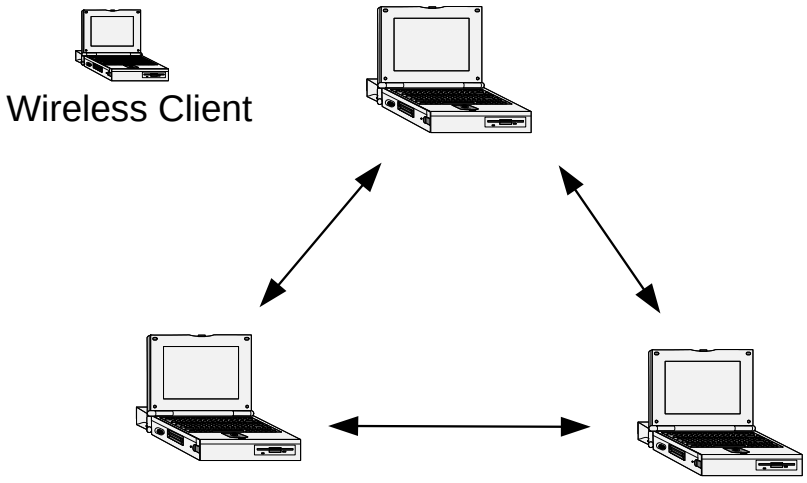

Figure 3.1: Peer-to-Peer

When communication is going through an access point, several access points can be connected together either with cables or by radio. This way the access

point can support hand-over and the service area can be extended.

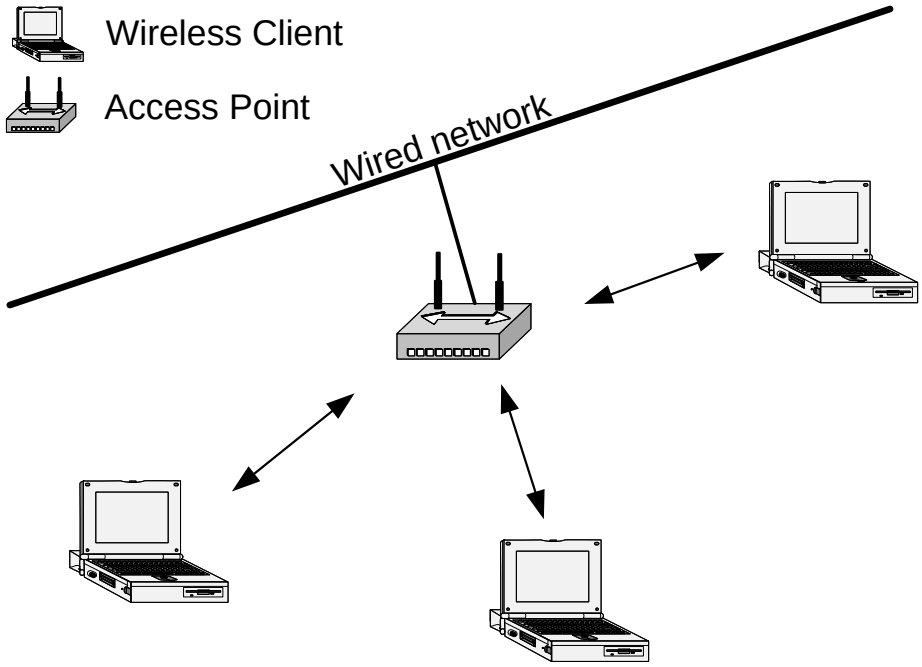

Figure 3.2: Access Point

With Peer-to-Peer devices are not dependent on an access point and two (or more) devices can create their own ad-hoc network without any existing network infrastructure.

3.1.3 Coexistence

Because many of the wireless technologies use unlicensed frequency bands, they are subject to interference from:

- Devices using other wireless technologies utilizing the same frequencies
- Devices using the same technology
- Microwave ovens

There have been many discussions about interference between Bluetooth and IEEE 802.11 (section 3.3) and how that will affect deployment of the technologies. One research paper from Ericsson [17] shows however that

Bluetooth *data* transfer will work acceptably in a dense IEEE 802.11b environment. But the same paper also shows that Bluetooth *voice* performance will suffer because unlike data, voice cannot be retransmitted.

The IEEE 802.11b devices will also work acceptably in a Bluetooth populated environment as shown in a study by Jim Zyren [34]. According to the study, the interference between IEEE 802.11 and Bluetooth increases as a function of range between the IEEE 802.11 access point and IEEE 802.11 terminal. Note however, that this study is based on theory, and Zyren concludes that testing in a lab environment will show whether his study is right or wrong.

Devices can also experience interference with other devices that use the same technology. With Bluetooth this means that a piconet might experience interference with other piconets. There is no interference within one piconet, because slaves are only allowed to transmit when the previous packet was addressed to them. But when two piconets are active side by side, interference among these piconets may occur. Bluetooth handles this with the frequency hopping, so interference between piconets is kept to a minimum. Other technologies, e.g. HIPERLAN2 (section 3.4) uses frequency allocation so interference between neighbor HIPERLAN2 devices (that are member of the same physical network) should not occur.

Microwave ovens can legally emit significant levels of leakage in the ISM (2.4GHz) band. This problem is discussed in a paper written by Buffler and Risman [7]. Their research shows that the leakage from microwave ovens is complex, but they also have some conclusions:

- The leakage of electromagnetic waves from microwave ovens is dependent upon the type of food being heated
- And more obviously, the leakage is highest in bigger, commercial microwave ovens, often found in restaurants

This shows that home networks will most likely not have problems, because home microwave ovens are used a relative short amount of time each day and the leakage in such ovens are small. Offices situated near cafeterias or restaurant might however face more interference, and in extreme cases wireless communication in such an environment might be impossible.

3.1.4 Security

While wireless radio devices make it easier for the user to be mobile, such devices are also, by nature, more insecure. In contrast to signals propagating through a cable, wireless radio signals do not stop at the receiver. With exception of infrared communication most wireless radio signals are also not directed at the receiver and signals can penetrate through walls. This makes

it important to implement strong encryption, to stop people outside the building monitor the transmissions on a wireless radio network.

Infrared communication (e.g. IrDA, section 3.2) is more secure because infrared light does not penetrate walls and furniture. To eavesdrop on a device using infrared light, you have to be in the same room. This means that one of the downside of not being fully mobile with infrared devices is a good thing when it comes to security.

3.1.5 Master/Slave vs. anarchy

In Bluetooth each device in a connection will have a master or slave role. The master/slave principle in Bluetooth introduces restrictions on which device that are allowed to transmit packets. This has some advantages, e.g. it is one device that can control the connections and make sure QoS restrictions are fulfilled.

Other specifications, such as IEEE 802.11, do not have such roles, and devices listen on the air for other transmissions before they transmit packets.

It should be noted that Bluetooth have higher layer protocols that hides the master/slave roles, and there are plans to incorporate QoS into IEEE 802.11.

3.2 Infrared Data Association (IrDA)

IrDA is a set of specifications [9] defined by the IrDA consortium. The specifications were published in late 1993, and later the suite of standards has been supplemented to include high-speed extensions for 1.152 Mb/s, 4.0 Mb/s and 16 Mb/s. This makes the transfer rate faster than that of Bluetooth, but the max range is also shorter (1.2m). As the name implies, IrDA is based on infrared light which limits the transmission to devices that are in line of sight. This also poses limitation on mobility, but, as noted in section 3.1.4, it also makes it more secure.

IrDA have much in common with Bluetooth, in fact the Bluetooth specification has borrowed the OBjEct eXchange (OBEX) protocol from the IrDA specification. In addition the vCard and vCalendar used to store business cards and calendar information are used by both technologies.

3.3 Wireless LAN (IEEE 802.11)

The IEEE 802.11 standards [18] specified by the IEEE 802.11 working groups is a set of standards defining the PHYsical (PHY) and Medium Access Control (MAC) layer (see figure 3.3) of a wireless local area network.

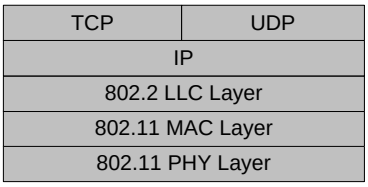

Figure 3.3: IEEE 802.11 with 802.2 LLC and TCP/IP

The IEEE 802.11 standards are part of the IEEE 802 Local Area Network (LAN) standards family, and on top of the IEEE 802.11 MAC and PHY layer we find the IEEE 802.2 Logical Link Control (LLC) layer. The LLC layer is the same for all IEEE 802 transport technologies, which are shown in figure 3.4, taken from Geier [13].

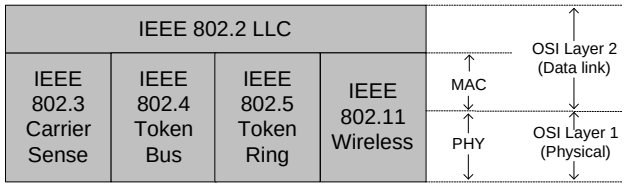

Figure 3.4: IEEE 802 LAN Standards family

The IEEE 802.11 standards consist of a common MAC layer and various PHY layers that use different transmission techniques and offer different bandwidth.

3.3.1 IEEE 802.11 MAC Layer

The MAC layer can operate in both peer-to-peer mode and access point mode, but access point mode provides hand-over and better coverage area. To avoid collisions with other devices the IEEE 802.11 uses a Carrier Sense Multiple Access / Collision Avoidance (CSMA/CA) scheme. In this scheme devices listen for traffic and only transmit packets if the channel is free.

We have summarized some of the key features of the IEEE 802.11 MAC services below. Note that QoS is not a part of these services, but an IEEE task group (IEEE 802.11e) is currently developing an extension to support QoS.

Service scanning

Service scanning is similar to Bluetooth inquiry and is used to detect information about the available networks. This information is needed to connect to a network and can be obtained with either *passive* or *active scanning*.

With passive scanning the device listen for a period of time to detect the information being sent between devices on the network. If passive scanning fails, the device can use active scanning and send out a *Probe*. Networks that receive the probe will send out a *Probe Response* with the information that is necessary to connect to this network.

Authentication/Encryption

In IEEE 802.11 two authentication methods are available: *Open System* and *Shared Key*. Open System authentication is not secure, because a device is allowed to connect to all devices that are set to use Open System authentication.

The Shared Key authentication is more secure, because a shared key have to be entered into all participating devices. Devices that do not know the shared key will not be able to connect to other devices.

Encryption in IEEE 802.11 uses the Wired Equivalent Privacy (WEP) algorithm. The WEP algorithm uses secret keys to encrypt messages over the wireless link, after authentication has taken place. Flaws have recently been found [11] in the RC4 algorithm used by WEP, and further studies [29] have proved that using WEP alone is not secure. The research concludes that IEEE 802.11 access points should be placed outside firewalls, and higher level encryption such as Secure SHell (SSH) should be used.

Power Management

An IEEE 802.11 device can enter a *sleep mode* to conserve power. When a device is in sleep mode the access point will buffer packets addressed to the sleeping device. The access point is therefore necessary, and power management is not available in peer-to-peer mode.

3.3.2 IEEE 802.11 PHY Layer

The IEEE 802.11 specification specifies three different PHY layers (see table 3.1).

The FHSS PHY layer uses frequency hopping, like Bluetooth, but with a much slower hopping rate, whereas Direct Sequence Spread Spectrum (DSSS) PHY uses direct sequence. All these techniques are however only of historic interest, because the extensions described in the next sections provides a much better transfer rate.

3.3.3 IEEE 802.11b PHY Layer

The original specification, as described in the previous section, has a maximum transfer rate of 2 Mb/s. The IEEE 802.11b task group has come up

	Bandwidth	Modulation	Frequency
FHSS	1-2 Mb/s	Gaussian Frequency Shift Key (GFSK)	2.4GHz
DSSS	1-2 Mb/s	Differential Binary Phase Shift Key (DBPSK) / Differential Quadrature Phase Shift Key (DQPSK)	2.4GHz
IR	1-2 Mb/s	16/4 - Pulse Position Modulation (PPM)	infrared

Table 3.1: IEEE 802.11 PHY Layer

with a different modulations scheme using 8-chip Complementary Code Keying (CCK). They have released a specification [18] that extends the IEEE 802.11 DSSS PHY and provides transfer rates up to 11 Mb/s. This PHY is referred to as High-Rate DSSS (HR/DSSS) and can coexist with traditional DSSS devices.

3.3.4 IEEE 802.11a PHY Layer

IEEE 802.11a is another high speed extension to IEEE 802.11 that uses Orthogonal Frequency Division Multiplexing (OFDM) and the 5GHz band to provide transfer rates up to 54 Mb/s. This standard is, however, not yet in widely use.

3.4 HIgh PErformance Radio LAN 2 (HIPERLAN2)

The HIPERLAN2 specification was published by ETSI in May 2000. HIPERLAN2 operates in the 5GHz portion of the radio spectrum (see table 3.2) and are therefore supposed to work seamlessly with Bluetooth. The features from the specification are given in [20] and summarized here.

Europe	5.15-5.35 GHz and 5.470-5.725 GHz
USA	5.15-5.35 GHz and 5.725-5.825 GHz
JAPAN	5.15-5.35 GHz (under discussion)

Table 3.2: Frequencies allocated by HIPERLAN2

3.4.1 Topology

The HIPERLAN2 topology normally consists of a wired network, access points and mobile terminals (as described in section 3.1.2). Multiple access

points can be connected together and hand-over is supported to extend the coverage area.

A direct mode of communication between mobile terminals (peer-to-peer) is also in development, but access points are believed to be the preferred method of communication.

3.4.2 Connection-oriented

HIPERLAN2 connections are time division multiplexed over the air interface and uses signaling to establish connections (which makes it connection oriented).

3.4.3 Quality of Service (QoS)

QoS are supported to ensure that critical data is transferred within the given limits. Examples are guarantees to bandwidth, delay, jitter and bit error rate. The support of QoS makes HIPERLAN2 suited for time critical applications like voice and video, but also data transfer, where it exist options to prioritize certain connections.

3.4.4 High-speed transmission

HIPERLAN2 supports data rates up to 54 MBit/s on the physical layer and 25 MBit/s on layer 3. To achieve this it uses OFDM to transmit the signals through the air. OFDM is suitable inside buildings, where the transmitted radio signals are reflected many times before they reach the receiver.

3.4.5 Frequency allocation

The Global System for Mobile communications (GSM) system also has access points (base stations) and mobile terminals like HIPERLAN2. To minimize interference between neighboring base stations, GSM companies must manually go through with frequency planning. In a HIPERLAN2 environment such frequency planning is not necessary, because the access points have automatic frequency allocation. The access points listen for neighboring access points and other nearby radio sources and selects radio channels based on this. In this way interference with other radio device is kept as low as possible.

3.4.6 Security

Authentication is supported in HIPERLAN2 to give the access point the possibility to deny unauthorized access to the network. The authentication process also ensures the mobile terminal that it is connecting to a valid network.

When authentication is performed, traffic can be encrypted to make eavesdropping on ongoing connections harder. The encryption uses the Data Encryption Standard (DES) and triple-DES (encrypted using DES three times) [21]. DES have been broken once [12], but this was done by 100.000 machines working together over the Internet. DES and triple-DES is thus considered secure for normal corporations, although a new encryption standard, Advanced Encryption Standard (AES) is currently under development to replace DES.

3.4.7 Mobility

HIPERLAN2 support mobility (hand-over) between access points. The mobile terminals will always choose the access point with best signal to noise ratio. When a hand-over occurs, all new connections will be moved to the new access point, but some packets may be lost.

3.4.8 Power save

To help the devices use less power, there are mechanisms to let the mobile terminals sleep for a give amount of time. This sleep period is initiated by the mobile terminal, and the mobile terminal will not listen for packets during the sleep. After the sleep the mobile terminal will search for a wake up indication from the access point. If a wake up indication is not received, it can go back to sleep.

3.5 HomeRF

The HomeRF Working Group was formed in March 1998 to specify a low-cost alternative for wireless home networking. They have released the Shared Wireless Access Protocol (SWAP) which supports voice and data traffic. Voice is supported by using the same techniques as Digital Enhanced Cordless Telecommunications (DECT), while the data part uses the FHSS part of IEEE 802.11. The result is shown in figure 3.5.

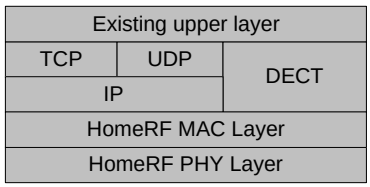

Figure 3.5: SWAP Stack

SWAP is designed to operate in the 2.4GHz ISM band, the same band as Bluetooth. The main applications are telephony and Internet sharing, but

the HomeRF Working Group use the term Home Area Network (HAN) in their advertising in contrast to Personal Area Networking (PAN) used by the Bluetooth SIG. The max range of devices based on the swap specification is 50m.

HomeRF devices can operate both with an access points and in peer-to-peer mode, but an access point is needed to support voice transmission. At this time data rates are max 1.6Mb/s, but products supporting data rates up to 10Mb/s is promised in the near future.

SWAP uses the blowfish [27] algorithm for encryption. Blowfish was designed in 1993 by Bruce Schneier and can be used as a drop-in replacement for DES. The algorithm is considered secure and has not yet been broken.

3.6 SPIKE

SPIKE [19] is a relative new technology and the specification will look familiar to those with knowledge of the Bluetooth technology. At this time SPIKE operates on the 900MHz ISM-band, but it is specified to also work at the same portion of the ISM-band as Bluetooth (2.4GHz).

The specification of SPIKE is not open as that of Bluetooth; you must license the technology to get knowledge of the internals of the specification. In addition SPIKE was developed by one company, Eleven Engineering, whereas Bluetooth was co-developed by several major companies.

Usage Scenarios

SPIKE was primarily developed for the gaming industry, as a cable replacement between a game console and a control box, but the usage is not limited to this. The technology can also be used as a LAN gateway (Master/slave), a MP3 server (Broadcast) or as a way of Peer-to-Peer communication as the usages models in figure 3.6 shows.

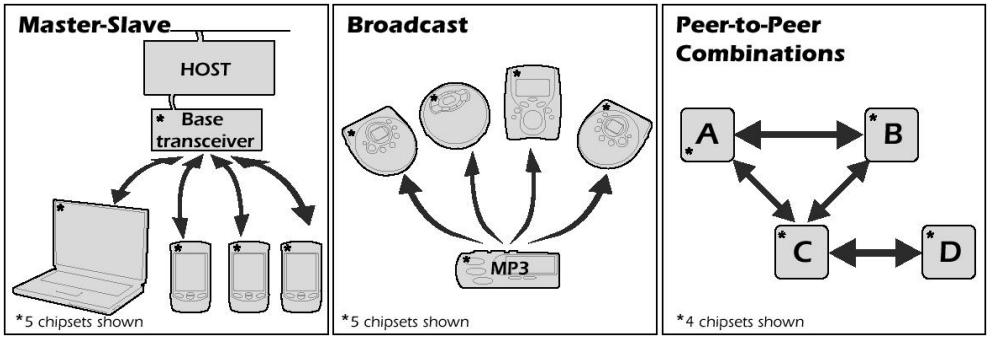

Figure 3.6: SPIKE - Usage Models

3.7 Short range wireless - present and future

Of the technologies presented here, on of the most successful at this time is IrDA which is built into most high-end cellular-phones, handhelds and laptops.

IEEE 802.11b networks are starting to be more common, especially in office environment with dynamic network infrastructure where wireless networks limits the need for re-cabling.

In the future there will probably be a demand for higher speeds, and HIPERLAN/2 or IEEE 802.11a might be the winners.

Among all these short-range wireless technologies Bluetooth may find its niche into battery powered devices like cellular phones and handheld computers. Laptops might also benefit from Bluetooth if there is not a demand for high-speed transfer rates. All this will however depend upon whether device manufacturers include Bluetooth into their products.

Chapter 4

Bluetooth network topology monitor

The goal for this thesis is to analyze various methods to get an overview of the Bluetooth network topology. To achieve this we consider the construction of a monitoring unit collecting the information needed to outline the Bluetooth network topology. In this chapter we will specify the requirements for the monitor, but first we review the main issues regarding Bluetooth network topology.

4.1 The Bluetooth network topology

Before we discuss how we will monitor the Bluetooth network topology it is essential that we have a clear view of the main characteristics of the topic. We will therefore elaborate on chapter 2 and present the parts of the Bluetooth specification that a Bluetooth network topology monitor must deal with.

4.1.1 Roles

At the basis of the Bluetooth network topology we have the master and the slave. A master can be connected to at most 7 slaves, and it is the master who decides when these slaves are allowed to transmit packets (in the context of this master).

4.1.2 Piconet

A master and the slaves connected to it are together called a piconet, and a piconet supports point-to-multipoint (figure 4.1b) communication where the master can send packets directly to all the slaves connected to it. The

piconet also supports point-to-point (figure 4.1a) which is a subset of point-to-multipoint communication.

Note that a device can not be master in more than one piconet, because a piconet is a group of devices hopping on a sequence defined by the clock and the BD_ADDR of the master [6].

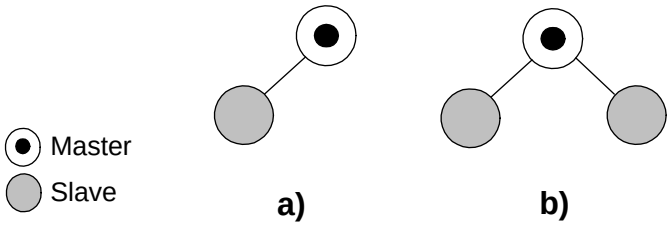

Figure 4.1: Piconets

4.1.3 Scatternet

A scatternet is a collection of at least two piconets connected together by one out of two ways:

- One device being *slave in more than one piconet* (figure 4.2a)
- One device being *master in one piconet and slave in another piconet* (figure 4.2b)

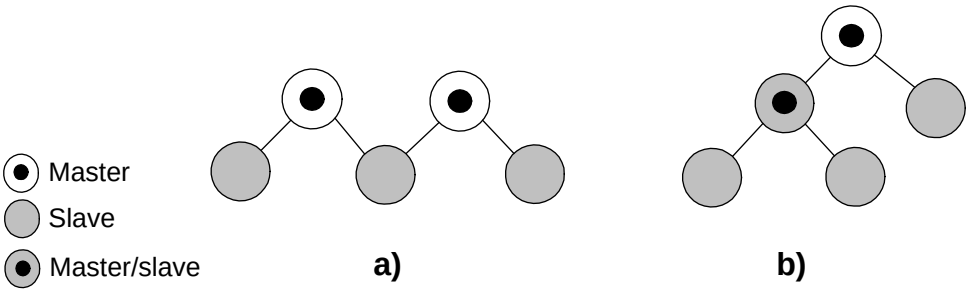

Figure 4.2: Scatternets

Devices that are connected to more than one piconet must time share its participation on the piconet, i.e. spend some time in each piconet. To achieve this, the device must also keep track of both hopping sequences (clock and frequency) so it knows how to synchronize after a piconet switch.

4.1.4 Changes in the topology

The Bluetooth piconets and scatternets can be highly dynamic, and the Bluetooth network topology may change often. Topology changes that can occur include:

1. Connection is tore down
2. Device is powered down
3. Device goes out of range
4. New connection
5. Master/slave switch

All these changes will lead to a different topology. In some cases (changes 1, 2 and 3) it may lead to a scatternet being partitioned into piconets. This is shown in figure 4.3 (from a to b) where the connection that holds two piconet together as a scatternet is tore down. The changes can also go the other way (from b to a) to connect two piconets to a scatternet.

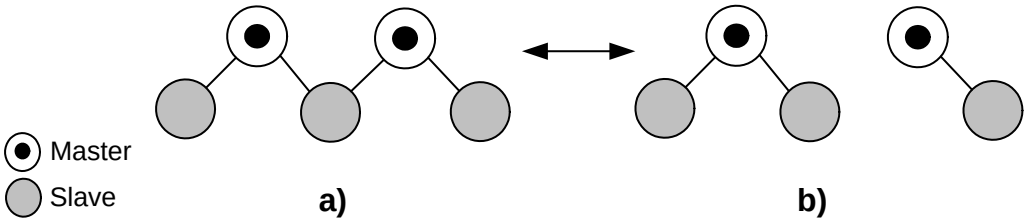

Figure 4.3: Changes in the topology

A master/slave switch can be initiated by both slave and master. One scenario is the gateway scenario explained in section 2.10.5. Note that a former master in a master/slave switch cannot hand over its old slaves to the new master. If the master has more than one slave before a switch, the additional slaves must be paged by the new master if they should be moved to the new master [6]. This operation is however not specified in the Bluetooth specification.

4.2 Requirements

We now have the theory we need and can specify the requirements for a Bluetooth network topology monitor. These requirements will be used when we later design the monitor, but first of all we state the requirements definitions we use.

The requirement definitions we use are taken from Sommerville [28] and are quoted here:

- **Functional requirements** describe statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.
- **Non-functional requirements** are constraints on the service or functions offered by the system. They include timing constraints, constraints on the development process, standards and so on.

In the next sections we specify the functional and non-functional requirements which we will use in the next chapters when we discuss the different solutions.

4.3 Functional requirements

The idea behind our Bluetooth network topology monitor is to have an application running on a device, obtaining the necessary information about the topology (figure 4.4). Please note that the monitor may be a part of the topology although it is illustrated as an external device in this figure.

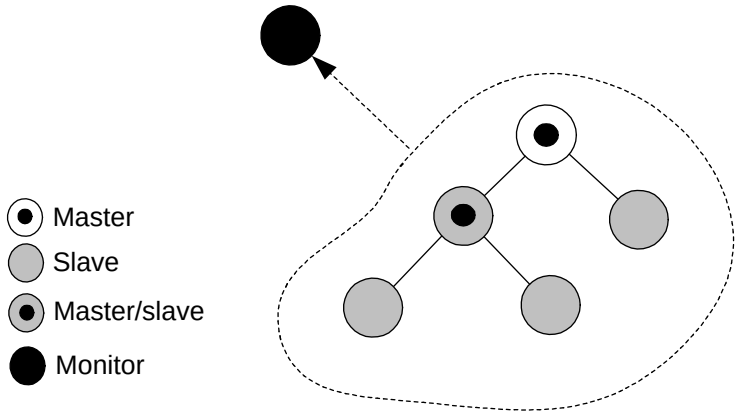

Figure 4.4: Monitoring unit

The monitoring unit illustrated in figure 4.4 must be able to detect the topology at start-up and detect the changes that can occur (described in section 4.1.4).

To ease the analysis we partition the monitor into two parts:

- Piconet Monitor

- Scatternet Monitor

A piconet monitor must be able to monitor a single piconet (i.e. a master and all the slaves connected to this master). The scatternet monitor will consist of several piconet monitors linked together to monitor a scatternet (figure 4.5).

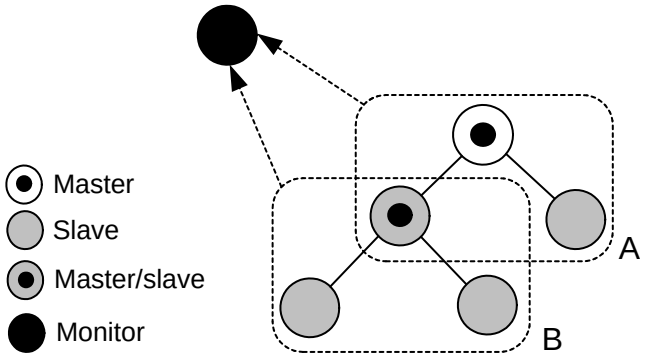

Figure 4.5: Piconet monitors linked together to monitor a scatternet

As we see it, this is a straightforward partitioning of the problem. There could also be other ways of implementing a scatternet monitor, e.g. without using our piconet monitor. This will however, not be discussed in this thesis, due to time constraints.

A piconet monitor will therefore be an essential part of any Bluetooth network topology monitor and a scatternet monitor can not be constructed without piconet monitors feeding it with information. A schematic figure that shows this is given in figure 4.6.

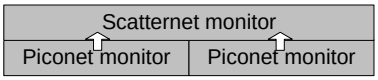

Figure 4.6: Piconet monitors feeding information to a scatternet monitor

4.3.1 Piconet Monitor

As stated earlier, a piconet monitor will monitor one piconet, i.e. a master and all the slaves connected to it. The monitor should be able to detect the BD_ADDR of all the devices and the different roles (master/slave) they have. Whether one of the devices in the piconet is member of other piconets is irrelevant in the context of a piconet monitor. Piconet A, in figure 4.5,

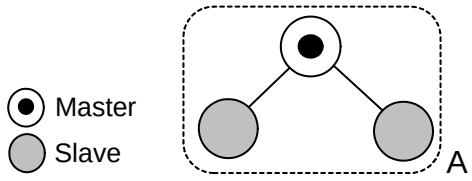

Figure 4.7: A piconet as seen by the piconet monitor

will thus be seen as shown in figure 4.7. Note that one of the slaves actually is master in another piconet, but the piconet monitor does not know this.

When a monitor starts up, it has no knowledge of the current topology. One of the first jobs to accomplish is thus to discover whether there exist a topology or not. If a topology exists, it must obtain information about which roles (slave or master) the devices in the topology have. A finite state machine that shows this is given in figure 4.8. The possible states are shown in **bold** and the valid state transitions are shown in *italic*.

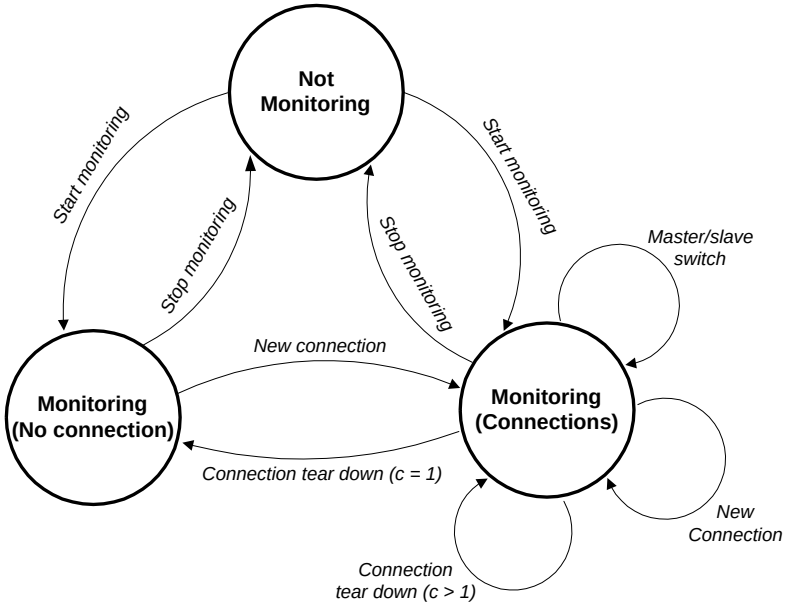

Figure 4.8: Finite state machine of a piconet monitor

A monitor process will always start up in the **Not Monitoring** state. From this state it will either go to the **Monitoring (No Connection)** state (if there are no connections) or to the **Monitoring (Connections)** state (if there are existing connections). This shows that the first step always will be to check whether there are existing connections or not.

The monitor may stop monitoring at any time by leaving one of the **Monitoring** states via the *Stop monitoring* transition.

From the **Monitoring (No Connection)** state there are only one possible transition except *Stop monitoring*, the *New connection* transitions which should be triggered when a new connection occurs.

If there are active connections, the monitor process will be in the **Monitoring (Connections)** state. In this state there can be a *Master/slave switch* or a *New Connection* and both transitions will lead back to **Monitoring (Connections)**. There can also occur a *Connection tear down* and this will lead to different states, whether there are one connection (c = 1, **Monitoring (No connections)**) or more (c > 1, **Monitoring (Connections**)).

All the transitions shown here must be taken into account when designing a piconet monitor. In most cases the transitions can fire events that can be captured, but an event-driven solution might not be available on the desired implementation platform. If it is not possible to subscribe to such events, the implementation must use timers and periodically check for changes in the topology.

4.3.2 Scatternet Monitor

A scatternet monitor will be constructed by linking several piconet monitors together. This means that a scatternet monitor is dependent upon services provided by piconet monitors and cannot operate without data from the piconet monitors.

To get a complete view of a scatternet there must be piconet monitors monitoring all the piconets. The data from these piconet monitors must be collected and analyzed by the scatternet monitor. This can probably be done in different ways and will be dependent upon various design decisions. The only requirement we specify is that data collected from all piconet monitors must be available at least one place for interpretation and presentation.

Interpretation

In the piconet monitor scenario interpretation of the collected data was not necessary; the application was only interested in the address of the attached slaves. With the scatternet monitor there is however a need to compare device addresses to interpret the topology information.

Let us take a look at the example topology in figure 4.9. To simplify the example we use simply one-letter names instead of the Bluetooth device address. This scatternet consists of two piconets (A,B,C and C,D,E) where C is slave in both piconets.

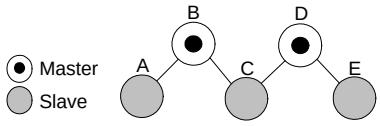

Figure 4.9: Example scatternet 1

If data from both piconets are collected, we will have a table that look like table 4.1.

Master	**Slaves**
B	A **C**
D	**C** E

Table 4.1: Monitor information of scatternet example 1

From such a table we can see that 'C' is a slave in both piconets and hence the piconets must be connected together via 'C'.

A similar deduction could be done if a device is master in one piconet and slave in another piconet (figure 4.10). This scatternet consists of two piconets (A,B,C and B,D,E) where B is master in the first piconet and slave in the second.

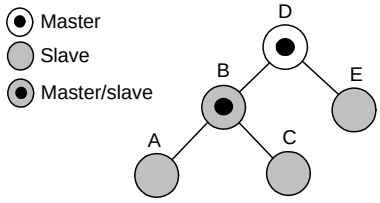

Figure 4.10: Example scatternet 2

Data collected from these piconet monitors will look like table 4.2.

Master	**Slaves**
B	A C
D	**B** E

Table 4.2: Monitor information of scatternet example 2

Here we see that 'B' is master in one piconet and slave in another. The analysis of the tables presented here is fairly easy with the human eye, but

our monitor must be able to do this automatically. We have therefore written some pseudo code to show how this interpretation could be performed by a computer.

Pseudo code for interpretation

The pseudo code we use should be fairly easy to understand for everyone with basic programming experience. We first present the terms we use in the pseudo code and explain what they mean.

```
FOR EACH item IN list:
```

Loop through *list* and extract each *item*.

```
IF item EXISTS IN list:
  <BLOCK>
```

Execute *<BLOCK>* if *list* contains the element *item*.

```
FOR EACH item IN list:
  IF value:
    NEXT
  <BLOCK>
```

Skip execution of *<BLOCK>* if *value* is true and iterate next loop in FOR-statement.

```
M
```

Set of all masters.

```
m.slaves
```

Set of all slaves connected to m.

The other terms used in the pseudo code should be straightforward, and it should be possible to understand the pseudo code for readers with some programming experience.

The pseudo code to interpret the topology data is given in table 4.3. Please note that the line numbers only are used to reference the details in the pseudo code and are not part of pseudo code. We will here give some explanation to the code.

The pseudo code includes two loops over all the piconets (line 1 and 2) and the first lines inside those loops (line 3 and 4) are used to make sure we only

compare different piconets (e.g. not compare one piconet to itself). If line 5 returns true, we have found a direct link between two masters.

At line 7 we start a loop that iterates over every slave in a piconet. The slaves returned throughout the iteration are compared (line 8) to the slaves in another piconet to see if a slave is member in more than one piconet. If line 9 returns true, we have an indirect link between the masters via a slave that is a member in both piconets.

```
1. FOR EACH m1 IN M:
2.   FOR EACH m2 IN M:
3.     IF m1 == m2:
4.       NEXT
5.     IF m1 EXISTS IN m2.slaves:
6.       <Direct link from m1 to m2>
7.     FOR EACH s IN m2.slaves:
8.       IF s EXISTS IN m1.slaves:
9.         <Indirect link via slave from m1 to m2>
```

Table 4.3: Pseudo code to interpret topology data

This pseudo code is not at all optimized. In addition the code will return all indirect links from masters via a slave twice (one each way). It is thus only meant as a starting point which can be refined in a final implementation.

4.4 Non-functional requirements

As we see it, there are some development decisions that have to be taken early and the most important are what hardware and software we should use to develop the Bluetooth network topology monitor. We have therefore identified these possibilities:

- **Custom Hardware solution**
- **Bluetooth solution**

In the custom hardware solution we do not impose any restrictions on the hardware available, and a solution may require custom-built hardware to get information about the Bluetooth network topology. To approach this solution, we will need a thorough understanding of how Bluetooth devices utilize the air-interface.

In the Bluetooth solution we will restrict the solution to use standard Bluetooth devices. Information about the network topology must be obtained

using services provided by a Bluetooth stack. In this solution we do not need to know how Bluetooth devices utilize the air-interface. What we however need to do is to study what kind of network topology information we can extract from a Bluetooth stack.

We discuss the custom hardware solution in chapter 5 and the Bluetooth approach in chapter 6, both based on the requirements specified here.

Chapter 5

Hardware Monitor

As explained in chapter 4, the approach presented here should not impose any restrictions on the hardware used in the monitoring of the Bluetooth network topology.

Our basic idea is that with custom-built hardware and knowledge of the Bluetooth frequency utilization we can build a standalone Bluetooth network topology monitor. This standalone monitor will eavesdrop on the Bluetooth traffic and will not be a member of any piconet. It will only analyze the traffic sent between other devices. From the information gathered in this process it may be able to detect active connections and get an overview of the Bluetooth topology. This chapter serves as a discussion of what we need to construct such a device.

We will first discuss the implementation of a piconet monitor, which is the simplest case.

5.1 Piconet Monitor

Based on the requirements from chapter 4 we will try to design a hardware piconet monitor. When we design such a device, it is important to know how to eavesdrop on the frequencies that Bluetooth operates and what information that is available on the air. The process of monitoring a piconet in this way will thus consist of two steps:

1. Synchronize to the piconet
2. Analyze the traffic

Synchronization is discussed in section 5.2 and analysis of the traffic in section 5.5.

5.2 Synchronization

If we want to eavesdrop on a Bluetooth connection, it is necessary to know when packets start, as shown in figure 5.1.

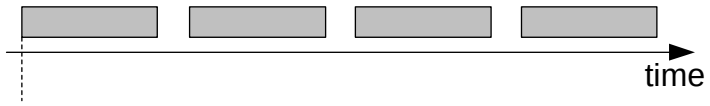

Figure 5.1: Synchronization in time

The synchronization in time is however not enough. Because Bluetooth uses frequency hopping (section 2.5.1), we must synchronize in both time and frequency, as shown in figure 5.2.

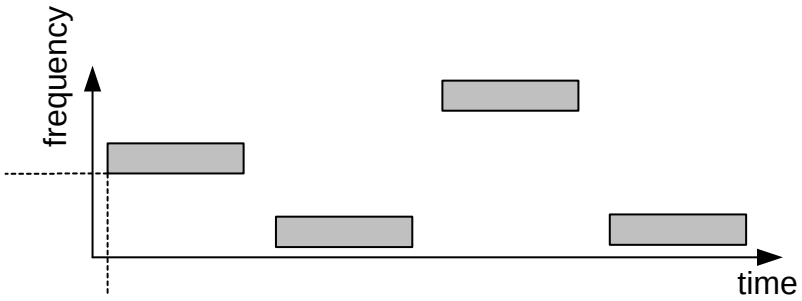

Figure 5.2: Synchronization in time and frequency

As explained in the section about paging (section 2.7.4), the frequency hopping pattern in a piconet is deduced from the BD_ADDR of the piconet master. In fact, the synchronization we try to achieve here has much in common with paging, but there is one essential difference. With paging the goal of the synchronization is to synchronize two devices that want to establish a Bluetooth connection. The goal of the synchronization in a hardware monitor is to synchronize to other devices *without establishing a connection.*

5.2.1 Synchronization with Bluetooth devices

To synchronize with a piconet we must obtain the piconet masters clock and the Lower Address Part (LAP) of the BD_ADDR. This information is available in Frequency Hopping Sequence (FHS) packets (section 2.6.4), and when a FHS packet is obtained from a device we have all information that is necessary to synchronize to the device.

When we have the information from the FHS packet, we can derive the frequency hopping pattern and synchronize our frequency hopping pattern

to that of the remote device. When the synchronization is fulfilled, the next hops will always be known to all participants, because the further hopping also is deduced from the BD_ADDR.

5.3 Protocol analyzer synchronization

To discuss various methods of synchronizing we have looked into some similar devices that are capable of analyzing traffic within a piconet. These devices are *Protocol analyzers* and a list and comparison of these devices are maintained by palowireless [24]. Protocol analyzers make it possible to follow and debug the communication over Bluetooth links and are a valuable tool for developers of Bluetooth products. With a protocol analyzer it is possible to capture all data sent between Bluetooth devices, and an example of a capture session is shown in figure 5.3. Note that while the goal with our network monitor is to monitor the various connections in a Bluetooth network topology, a protocol analyzers job is to monitor the *data flow* on those connections.

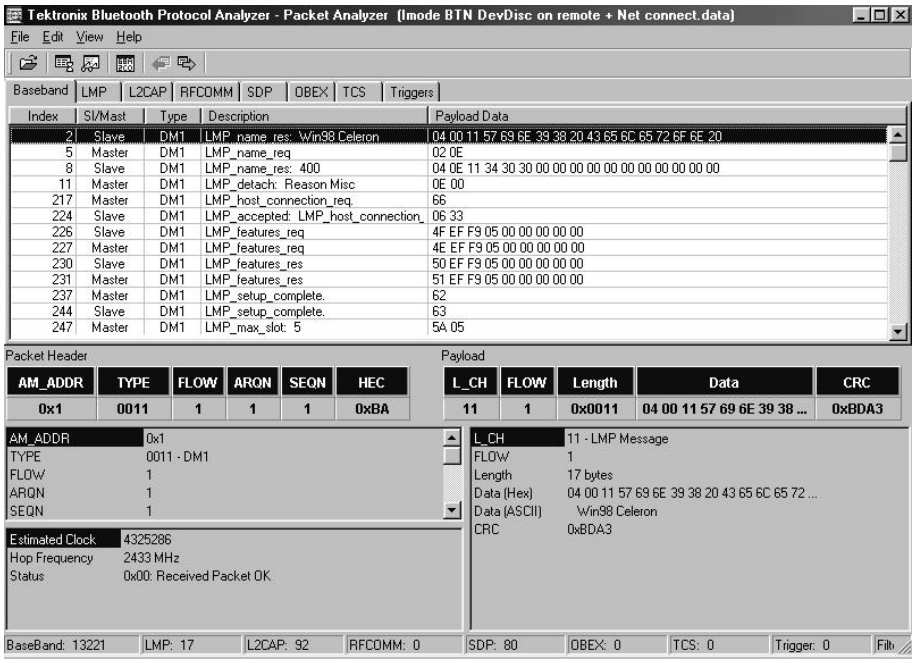

Figure 5.3: BPA100 Screenshot

Protocol analyzers operate in either *independent mode* or *piconet mode* and those that support independent mode are most interesting to us. Independent mode means that the device is capable of synchronizing to a piconet

and eavesdrop on the traffic without being a member of the piconet. Piconet mode is not interesting to us here, because in this mode the protocol analyzer will be a part of the piconet and will use standard Bluetooth synchronization. This bears a close resemblance to our software approach which we discuss in chapter 6.

The protocol analyzers use different techniques to synchronize to a piconet, and our aim is to reuse ideas from these devices when designing the synchronization of our piconet monitor.

Of the available protocol analyzers we have taken a closer look at the **Tektronix BPA100** [2], **CATC Merlin Bluetooth Protocol Analyzer** [3] and **Arca Wavecatcher** [32]. In the next section we will outline the specification of these devices, emphasized on how they synchronize to the piconet. This will be used to discuss whether we may use some of the same synchronization techniques in our Bluetooth network topology monitor.

Before transmission over the air can be captured the Protocol Analyzer has to synchronize to the piconet so it knows when the packets start and at which frequency they are transmitted. The synchronization is handled in different ways by the available products and we will try to summarize the possibilities here. This section is mostly based on a technical brief [31] from Tektronix, the producer of the BPA100 Bluetooth protocol analyzer. The technical brief gives a good overview of how synchronization are carried out on Tektronix' protocol analyzer. Arca and CATC have not released such documents for their protocol analyzers, but by looking at the options in their user manual we try to deduce how the synchronization are carried out.

5.3.1 Tektronix BPA100

The Tektronix BPA100 can operate in both the piconet mode and independent mode described in section 5.3.

To understand the synchronization process a thorough understanding of the Link Controller States (given in section 2.7) are required. BPA100 utilizes various link connection states to synchronize to a device, by executing the first steps in an inquiry or page sequence.

The BPA100 supports a subset of the standard Bluetooth link controller states (section 2.7) and two extra states as shown in figure 5.4. *Monitor Slave* and *Monitor Piconet*, the two new states, are states where the monitoring takes place. The subset of the standard states is used to reach the two new monitor states.

Common for all the synchronization methods used by the BPA100 is that the user has to have some knowledge of the existing or future topology. In

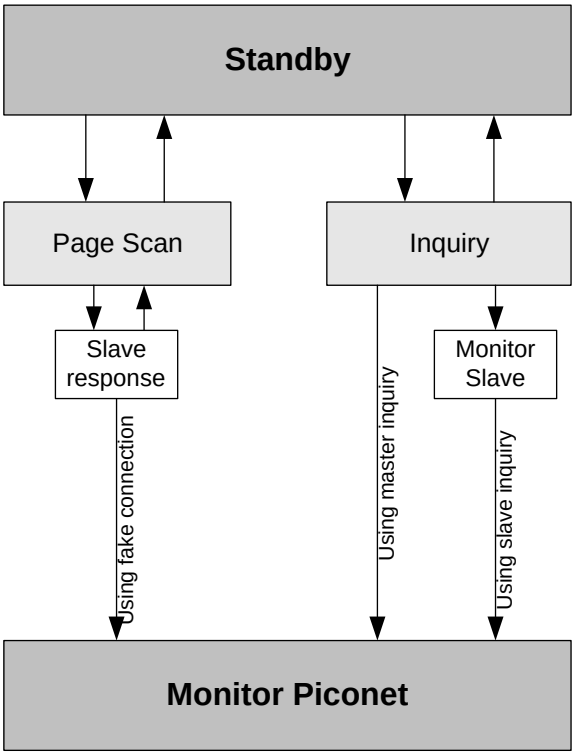

Figure 5.4: States used by BPA100

this section we will explain how the different synchronization techniques are carried out and what requirements they have.

Definition: In the following sections we use the term *target device* to refer to the device we want to synchronize to.

Synchronization using *master inquiry*

When synchronizing using master inquiry, a process similar to normal inquiry is carried out. Inquiry is usually used to get information about all nearby devices, more specific the devices that regularly enter inquiry scan mode. When a device enters inquiry scan mode and receives an inquiry packet, it responds with a FHS packet. As described earlier, the FHS packet includes all information that is needed to synchronize to a given device.

Like normal inquiry the master inquiry carried out by the BPA100 will receive a FHS packet from all devices that are in inquiry scan mode. There is however only one FHS packet that the BPA100 is really interested in, the one from the device it wants to synchronize to. All other FHS packets will be discarded.

Requirements for master inquiry synchronization

- Target device has to be the master of an existing or future piconet
- Target device must be discoverable, e.g. it must enter the inquiry scan mode at given intervals

Synchronization using *slave inquiry*

This synchronization method is similar to the master inquiry method, but with this method the BPA100 first connects to a device that is going to be the slave in a future piconet. A figure that shows this process is given in figure 5.5. When explaining this method of synchronization, we will refer to this figure.

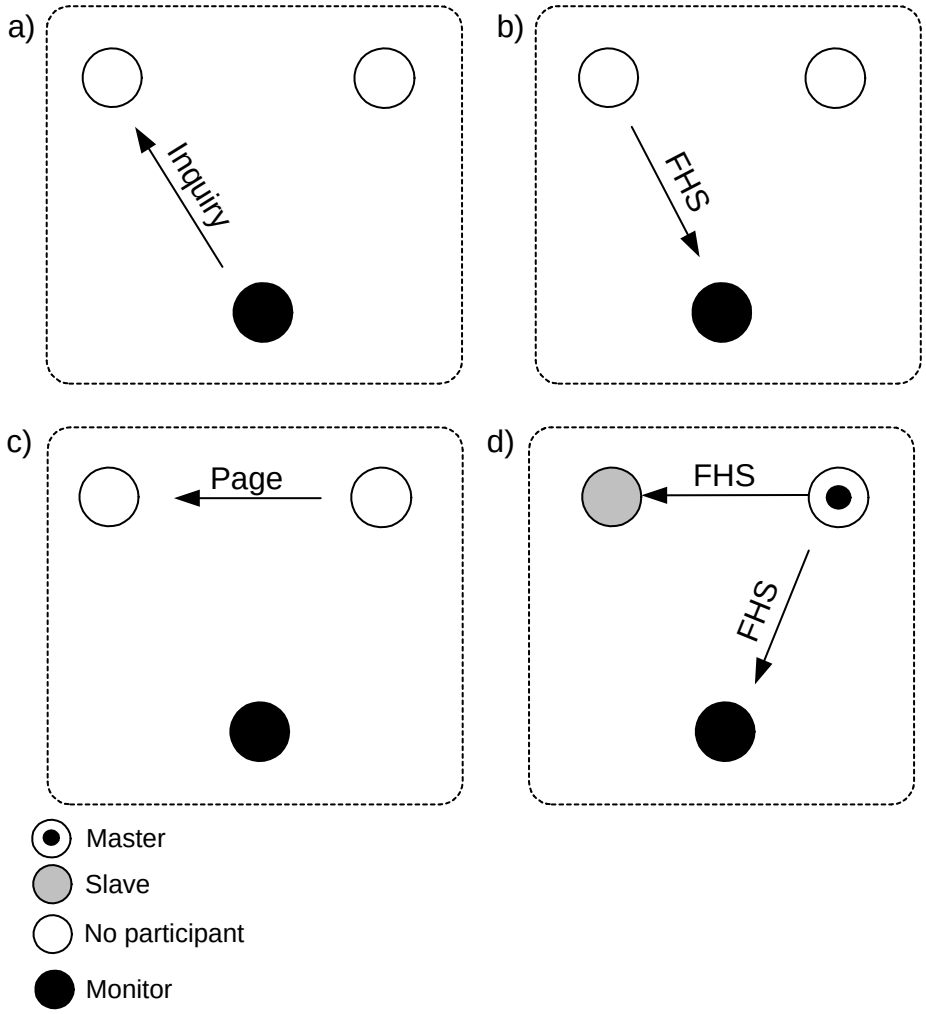

Figure 5.5: BPA100 - Synchronization using slave inquiry

At first a process like normal inquiry is carried out (a), to obtain the FHS packet of the target device (b) which is going to be a slave in a future piconet. With this FHS packet the BPA100 can synchronize to the device and monitor the traffic. When the target device is being paged by the master of the future piconet (c), the FHS packet from this master can be captured (d). The BPA100 can then resynchronize with the master in the now established piconet.

Requirements for slave inquiry synchronization

- Target device has to be the slave in a future piconet
- Target device must be discoverable, e.g. it must enter the inquiry scan mode at given intervals

Synchronization using *fake connection*

In this synchronization method the BPA100 pretends to be the slave device that a future master is paging. When the paging device sends the page message to establish the piconet, the BPA100 responds to the page before the response from the real slave. The future master will then send a FHS packet to the BPA100 and ignore the late arriving page response from the real slave.

In a normal page sequence the future slave is supposed to acknowledge the masters FHS packet to complete the connection. Because the BPA100 does not do this, the future master will redo the paging after a timeout. This time the BPA100 will not interfere in the paging process, the connection can be established and the BPA100 is synchronized.

Requirements for fake connection synchronization

- Target device has to be the master in a future piconet
- Target device must be able to act as a master in a piconet, and the master must be set up to page a given device.

Synchronization using *fake connection response*

The synchronization method called fake connection response makes it possible to synchronize to an existing piconet, but the master in the piconet must support point-to-multipoint operation. To make this synchronization possible the master in a piconet must page the BPA100 while it has an active connection to another device. The BPA100 is here set to be in the page scan state and will receive the FHS packet sent from the master in the page sequence.

When the BPA100 has received the FHS packet, it will not respond, and after a timeout the master will resume its existing connections. The BPA100 has now received the FHS packet and can monitor the piconet.

Requirements for fake connection response synchronization

- Target device has to be the master in an existing piconet
- Target device must support point-to-multipoint operation
- Target device must be set up to page the BPA100

5.3.2 Arca Wavecatcher

Arca has not released detailed specifications of the synchronization like Tektronix have done. From the Arca Wavecatcher Manual [30] we can however read about the different options available when monitoring in independent mode. We compare these options we with the Tektronix' BPA100 options and try to find similarities.

Page slave and monitor (non-existing piconet)

This synchronization is much the same as BPA100's slave inquiry where a connection to a slave is established and the FHS from the master is captured when the piconet is established. But the BPA100 uses inquiry to accomplish this whereas the Arca Wavecatcher uses paging.

Requirements for page slave synchronization

- Target device is slave in a future piconet
- The slaves BD_ADDR in a future piconet must be known
- Target device must be connectable, e.g. it must enter the page scan mode at given intervals

Inquiry master and monitor (existing piconet)

From the manual this synchronization method seems to be the exact same as the BPA100 master inquiry synchronization, except one difference. The BPA100 master inquiry works with future and existing piconets, but the Arca Wavecatcher seems to only work with existing piconets.

Requirements for inquiry master synchronization

- Target device has to be master in an existing piconet
- The masters BD_ADDR in a piconet must be known
- Target device must be discoverable, e.g. it must enter the inquiry scan mode at given intervals

5.3.3 CATC Merlin Bluetooth Protocol Analyzer

As Arca, CATC has not released detailed specifications of how their device synchronizes with a piconet. Because of that, this section is based on the CATC Merlin Bluetooth Protocol Analyzer manual [8] which we discuss and compare with the Tektronix BPA100.

Establish a new piconet and monitoring it

According to the Merlin manual, the user enters the address of the master and slave of a future piconet to be monitored. After the addresses are entered, the piconet can be established and the protocol analyzing will begin.

It is not explained why the Merlin needs both the master and slave address. Both the BPA100 and the Arca Wavecatcher is able to synchronize to a future piconet by only knowing the BD_ADDR of the future slave.

Requirements for synchronizing to a future piconet

- Target devices are slave and master in an existing piconet
- The BD_ADDR of the target devices must be known
- Target slave device must be discoverable, e.g. it must enter the inquiry scan mode at given intervals

Monitoring an existing piconet

When a piconet is already established, this method of synchronization is applicable. The user have to enter the BD_ADDR of the piconet master and know if the master responds to inquiries or can be connected to more than one slave.

For the user it is most convenient if the master periodically enters inquiry scan, because then the user do not have to do additional steps to complete the synchronization. By sending out an inquiry request the master will send its FHS packet to the Merlin.

If the master does not enter inquiry scan, but can connect to more than one slave, the user must select this option and have the master connect to the Merlin Protocol Analyzer. In this connection attempt the Merlin will receive the masters FHS packet, but does not respond to the connection request. The connection attempt will time out and the Merlin is synchronized to the piconet.

If the master never enters inquiry scan mode and can not connect to more than one slave, this synchronization method can not be used.

Requirements for synchronizing to an existing piconet

- Target device has to be master in an existing piconet
- The BD_ADDR of the target device must be known
- Target device must either periodically enter inquiry scan or be able to have more than one slave connected

5.3.4 Which synchronization method to use?

As presented here, the protocol analyzers have several synchronization methods to use and more than one may be applicable in a given context. The reason for the many synchronization methods are however that the protocol analyzers are targeted at Bluetooth device developers, and at an early design stage all the link controller states might not be implemented.

Another issue is the timeframe of the various synchronization methods. Synchronization methods utilizing inquiry will have a worst case timeframe of 10.24s in and optimal environment (no interference and no active SCO-links). Synchronization methods utilizing paging however will have a worst case timeframe of 2.56s. In addition to the timeframe issue some of the synchronization methods require additional input from the user. One example is the fake connection synchronization used by the BPA100, where the user must set up the target device to page a known device.

5.4 Protocol analyzers as a monitor

The three protocol analyzers described in this chapter are not topology monitors. Their monitor capabilities do not cover analyzing connections, they monitor all traffic sent within piconet and are capable of distinguish and analyze various Bluetooth packets.

Because of the synchronization, it can only monitor packets sent within one piconet, and the user are required to have knowledge of the topology in advance, i.e. some device address must be entered by the user. Our conclusion is that the synchronization methods used by the protocol analyzers are not applicable to a Bluetooth network topology monitor.

5.5 Analysis of the traffic

Although we did not succeed to find a general way to synchronize to a piconet, we will discuss the analysis of captured traffic.

To analyze the traffic we assume that synchronization have been carried out. This could appear to be an unrealistic assumption, because we have already concluded that such synchronization could not be carried out by

our monitor. The information that follows here are however useful in the discussion of eavesdropping on Bluetooth connections.

If we were able synchronize to a piconet, we could tune our radio to capture all traffic. But what information is available in the captured packets? To take a closer look at that we refer to section 2.6.4, where we have explained various parts of the Bluetooth packets. In this section we will discuss what this means to the hardware version of our Bluetooth network topology monitor.

5.5.1 Normal traffic

With normal traffic we mean a standard communication link between two Bluetooth devices. All packets on such a link will have an access code and a header, followed by the payload (figure 2.4). The only addressing information available here can be found in the header, but the address revealed here is not good enough for us. With our monitor we want information about the Bluetooth network topology, and we will then need the 48 bit BD_ADDR of all the devices in the network topology. A piconet however, uses only the 3-bit AM_ADDR to identify the participants. The AM_ADDR is a local address, and we can not use it to find out if a device is a member of more than one piconet.

5.5.2 Connection establishment

Although the BD_ADDR is not revealed in normal Bluetooth communication, it is accessible when exchanging FHS packets. The FHS packet is sent from devices in the page scan (section 2.7.4) and inquiry (section 2.7.3) state and when a master/slave switch occurs. This is in fact the only times the BD_ADDR is revealed in Bluetooth packets.

5.6 Hardware solution - conclusion

The synchronization methods used by various Protocol Analyzers shows that it is possible to synchronize to a Bluetooth piconet without active participation. But it also shows that such synchronization requires some information about the topology in advance, which is a too strict requirement for our topology monitor.

In addition section 5.5.2 show that the BD_ADDR is not revealed in normal Bluetooth communication. This means that we would have problems both with synchronizing and the analysis of the captured data. Analysis of the capture data showed that the BD_ADDR are revealed in FHS packets, but FHS are only exchanged between devices during device discovery, paging and master/slave switch. If a monitor should be dependent upon capturing these FHS packets, the synchronization would be even harder. It would

also be complex detect an existing connection, because no FHS packets are exchanged in an ongoing connection.

We therefore conclude that it is not possible to construct a Bluetooth network topology monitor by using custom-built hardware.

Chapter 6

Bluetooth Solution

In our Bluetooth solution we are limited to the services provided by a standard Bluetooth device and a Bluetooth stack. It is not important what profiles the Bluetooth stack supports, because all information about connections are provided by the Generic Access Profile and all Bluetooth stacks support this profile.

In this chapter we will try to design a software monitor based on the requirements specified in chapter 4.

6.1 Piconet Monitor

As with the hardware approach, we will first consider a simple monitor application, monitoring the connections within one piconet.

6.1.1 Requirements

The requirements for the application are given in section 4.3.1.

6.1.2 Design

The hardware device we tried to design in chapter 5 had to synchronize to the piconet when trying to detect connections. With access to a Bluetooth stack this is not necessary, because the synchronization will be implemented in the stack. A Bluetooth stack must in some way store information about the active connections, and hopefully it will offer services that give information about connections to an application using the stack. Applications can then go straight to the stack and identify the piconet topology.

We have identified the following questions that should be answered when designing the piconet monitor:

1. In which devices can the monitor be implemented, and where should it be implemented?

2. When should the changes be detected?

   (a) Periodical?

   (b) Upon change?

3. How should the network topology be presented?

These questions are further refined and answered in the following sections.

Location of the monitor

In the hardware solution in chapter 5 the goal was a standalone device. With the Bluetooth solution we need a fixed connection to the Bluetooth stack, and the device will probably be a part of the Bluetooth topology that we are monitoring. One of the first issues to clarify is thus in which device the monitor should be located, in the master or one of the slaves. Because of how Bluetooth is designed, a slave in a piconet is only aware of one connection, the connection to the master. The master will have information of all connections in the piconet and the monitor should thus be located there (figure 6.1).

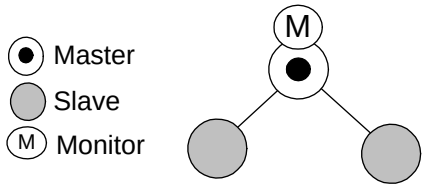

Figure 6.1: Monitoring one piconet

In the previous paragraph we concluded that the monitor should be implemented as an application on the master. The problem is however, that Bluetooth networks are expected to be created in an ad-hoc fashion, and we do not know whether devices will be masters or slaves before connections are made. Some devices might be designed in such a way that they will never initiate connections, just accept them, and never allow a master/slave switch. In these devices we do not need to, and possibly can not, implement the monitor application. All the other devices must however, have a monitor application, because it might be needed. The monitor can then run if the device is a master and stay idle if its current role is slave.

Detection of change

With the monitoring application running in the master the job of the application will be to extract the connection information from the Bluetooth stack. This extraction could be done either by pulling the information from the Bluetooth stack or, if the stack supports it, let the stack push new information to the application.

Whether the extraction of connection information is done by push or pull, the monitor should react to the state transitions specified in section 4.3.1, which represents changes in the topology. The transitions (topology changes) could be implemented as events, where the application will be notified each time a transition occurs. This corresponds to the push-model explained in the previous paragraph.

If not events are supported by the Bluetooth stack, the application must check for changes at reasonable time-intervals (i.e. pull information from the Bluetooth stack). The interval must not be too short, because this will generate unnecessary load on the Bluetooth device. However, the interval must not be too large either, because this can cause topology changes to be missed.

Presentation of information

The information that should be presented from the piconet monitor must at least include the information shown in table 6.1.

Description	**Type**
Master	BD_ADDR
Slaves	BD_ADDR 1, ..., BD_ADDR N

Table 6.1: Information gathered by a piconet monitor

This information includes the BD_ADDR of the master and the BD_ADDR of all the slaves connected to the master. The information should be available at least as raw data that could be fetched from other applications, e.g. the scatternet monitor in the next section. Optionally we could also present the information in a human-readable format suitable for viewing on screen. This could be used as a simple topology monitor, monitoring only piconets.

6.2 Piconet monitor - implementation

Based on the design presented in section 6.1.2 we have implemented the piconet monitor. The development environment and the final application are presented here.

6.2.1 Development Environment

In the application development we used two Motorola Bluetooth PCMCIA cards. The Motorola cards are produced by Digianswer [1] that also produces Bluetooth PCMCIA cards for Toshiba, IBM, DELL and NEC. A software suite with drivers and software that supports the standardized Bluetooth profiles (up to V1.1) [15] are bundled with the Motorola cards. In addition the software supports the PAN profile that is not yet released to the public.

The Motorola Bluetooth PCMCIA cards are also bundled with a Software Development Kit (SDK), currently in version 1.09. With the SDK it is possible to communicate with the Motorola Bluetooth PCMCIA card and develop Bluetooth applications. The SDK consists of an Application Programming Interface (API) to COM objects that communicate with the PCMCIA card. Instead of communicating directly with the PCMCIA card the developer communicate with the COM objects. A more detailed description of the API is available in the "Programmers Manual" [10]. The "Programmers Manual" includes both an introduction and a reference to C++ programming with the SDK.

In principle any programming language that supports communication with COM objects (Borland Delphi, Borland C++, MS Visual C++) may be used. Most examples found on the Digianswer website use MS Visual C++, although a few Borland Delphi examples exist. In addition the manual that covers the usage of the SDK covers MS Visual C++ only. We thus used MS Visual C++ (version 6 and 7 beta) in the development, and the environment is schematically shown in figure 6.2.

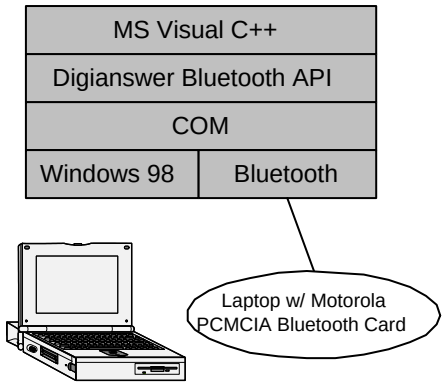

Figure 6.2: Development environment

The available methods in the Digianswer Bluetooth API are listed in the

Bluetooth Software Suite "Programmers Manual" [10]. There are methods to control the **Link Manager** and the following profiles:

- Serial Port Profile
- LAN Profile
- DUN Profile
- FAX Profile
- OBEX Object Push Profile
- OBEX File Transfer Profile
- Ethernet (PAN) Profile
- Headset Profile
- RFCOMM
- Service Discovery Profile

These profiles cover all the profiles in the Bluetooth Specification. The link manager is used to control link-related operations, e.g. connection establishment, master/slave switch and device discovery. However, one thing is missing in the Digianswer software; scatternet support.

As a Visual C++ reference, we used Visual C++ 6 Bible [22]. The main focus of this book is user interface programming with Visual C++, but the book also covers win32 specific parts of the C++ language.

6.2.2 Goal of application

The goal of the application development was to explore the development environment and try to construct a simple piconet monitor based on the requirements from chapter 4 and the design decisions in section 6.1.2. Some design decisions were refined during the implementation as we got experience with the Digianswer Bluetooth API.

6.2.3 The application

The code listing of the application can be found in appendix B.

We will here give a brief summary of the various files in the application. The source code consists of files with the ".cpp" and ".h" suffix. Those familiar with C++ or C will know that files ending in ".h" are header-files with definitions and files ending in ".cpp" are implementation files. In the following list we only name the base name of the file and cover both header files and implementation files at the same time.

- **Monitor** This is the class that starts up the main dialog. The dialog starts up a new instance of the *CMonitorDlg* object, specified in the *MonitorDlg* files. We also set up the communication with the COM object that handles all further communication with the Bluetooth device.

- **MonitorDlg** In this class the main application (i.e. monitoring) is implemented. We also instantiate the eventhandler, timer and the mapping from the user interface to the application.

- **eventhandler** The eventhandler handles all events fired by the Bluetooth stack. When a connection is established or tore down, corresponding functions in *eventhandler.cpp* will be executed, to update the view of the topology.

- **BTAddress** BTAddress includes various functions that helps manipulate Bluetooth devices addresses (e.g. convert from a 48-bit internal format to a 17-byte human-readable format, i.e. AB:CD:EF:GH:IJ:KL).

- **stdafx** Specific includes that are used frequently, but changed infrequently. Among other things it includes the header files for the Digianswer Bluetooth API (COM).

We do not cover the design of the user interface here, because that is not an important aspect of the piconet monitor.

When the monitor starts up, and we have no slave-connections, the output will be like figure 6.3.

Figure 6.3: Screenshot of the Piconet monitor (No slaves)

The address of the local device is shown in the title bar, but we are not master in any connections so the list of slaves says "<NONE>". In figure 6.4 we see the output when our local device has two active slaves. The output of the piconet monitor will then show the address of the two slaves.

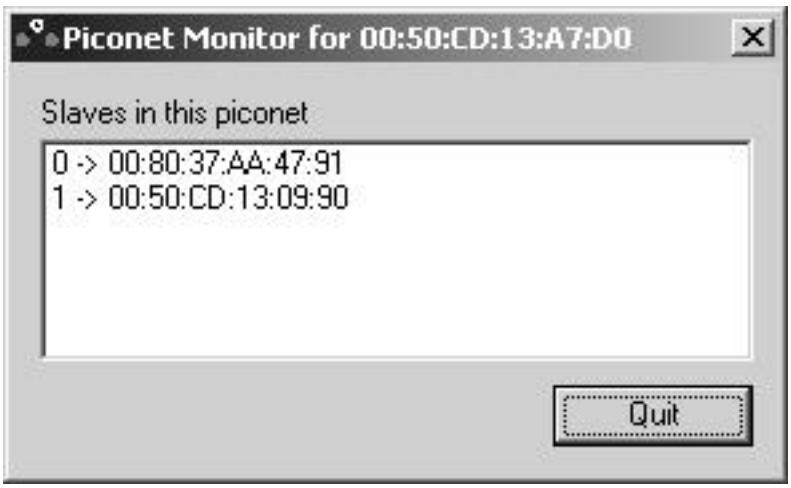

Figure 6.4: Screenshot of the Piconet monitor (2 slaves)

6.2.4 Comments to the implementation

As a starting point in the development process, we downloaded the examples from the Digianswer Bluetooth Software suite site [10]. The examples were more advanced than what we needed so we started from scratch and included just the code that we needed from the examples. From this process we got a thorough understanding of the API that was essential in the implementation.

Events and timer

The Digianswer SDK supports events that detect connection establishment and connection tore down. These events were used to update the topology information whenever such topology changes occurred. The implementation of these events can be found in *eventhandler.cpp*.

Master/slave switches did however not fire any events. To compensate for this we had to set up a timer with a 10s interval to detect a master/slave switch. The implementation of the timers can be found in *MonitorDlg.cpp*.

Connections

We choose to store the connection information in a BTPiconet struct that look like this:

```
struct BTPiconet
{
    // Constructor
    BTPiconet() {
        slaveCount = 0;
        master = NULL;
    }
    char * master;   // master BD_ADDR
    char * slave [7]; // BD_ADDR of the slaves
    int slaveCount;  // number of slaves in this piconet
};
```

This corresponds to table 6.1 from the design phase, but in addition we use a *slaveCount* integer to ease the manipulation of the struct.

6.3 Scatternet Monitor

In the previous section we successfully designed and built a piconet monitor that was able to monitor a single piconet. The requirements we specified in chapter 4 shows that a scatternet monitor should be built by extracting information from several piconet monitors and interpret the collected information. To achieve this we assume that we have piconet monitors running on each piconet. This mean that a scatternet monitor does not have to monitor connections, only collect information provided by the piconet monitors.

6.3.1 Assumptions and prerequisites

In a scatternet there are likely to be both devices that cannot be programmed, like headsets, and programmable devices like personal data assistants and notebooks. To achieve our goal we assume that all devices where the monitor application is supposed to exist are devices that give the opportunity to implement the application we describe.

Our goal will be to monitor an existing scatternet topology and consequently monitors should establish links between them on existing connections and not establish connections by themselves. If the monitor made new connections by itself the topology would change and that is not acceptable.

6.3.2 Design

The scatternet monitor could be seen as a distributed application where all piconet monitors are connected to a central network. We thus propose that the scatternet monitor should be implemented as a distributed application exchanging information about piconets. In this case it needs to be implemented in the nodes where the piconet monitors are running, either as an external application or as an extension to the piconet monitor.

Our main focus in the design phase will be discussing the challenges we are likely to meet and propose solutions where that is appropriate. We have not implemented the design we propose, because scatternet operation is not yet available in most development environments, including the Digianswer SDK.

Like we did with the piconet monitor, we have identified some questions that should be answered in the scatternet design phase:

1. In which devices can the monitor be implemented, and where should it be implemented?
2. How will the exchange of information between monitors be carried out?
3. How should the notifications be done?
   (a) Periodical?
   (b) Upon change?
4. How should the network topology be presented?

Some of these questions are identical to those discussed in the piconet monitor scenario. However, these questions have to be answered at a higher level, i.e. scatternet, not piconet. With the piconet monitors we had to deal with information retrieved from the Bluetooth stack, but with the scatternet monitor we have to deal with information retrieved from the piconet monitors.

The questions are discussed in the following sections and solutions are suggested where appropriate.

Location of the monitors

In the piconet scenario we placed the monitor in the masters, and as explained the scatternet monitor will be located in the same devices. In this way all masters will monitor their own piconet, but in the scatternet scenario this is not enough. We need to be sure that all monitors are connected to each other so they will have the opportunity to exchange information, and this is not naturally fulfilled in all scatternet topologies. In chapter 4 we saw two ways piconets could make up a scatternet. One possibility was that a slave in one piconet was master in another piconet. In figure 6.5 we see that in this case there are already direct connections between the monitors and the direct-connection requirement is fulfilled. The two monitors here can exchange information on the existing connection and both monitors will get a complete view of the topology.

In the other scatternet case, where one device is slave in more than one piconet, there is not automatically such a direct connection between the

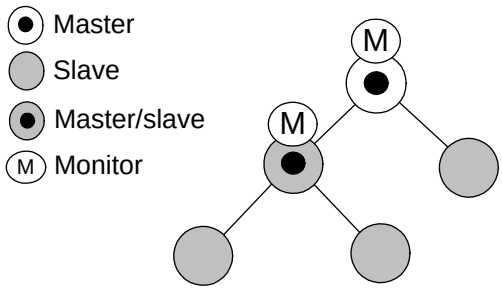

Figure 6.5: Monitoring two piconets

monitors in the masters. A simple approach to solve this would be to place a monitor in those slaves, which will give us an indirect monitor-link between the masters.

Although we could place a monitor in all the slaves that are connected to more than one piconet, they do not need full monitor capabilities. In figure 6.6 we see that we do not need any information from the slave to get a complete view of the topology. If we are able to compare information provided by the monitors placed in the masters, we would have enough information to present the complete network topology. The slave needs to forward all monitor information it receives to achieve this. If this can be done, the monitors can exchange information via the slave and be indirectly connected.

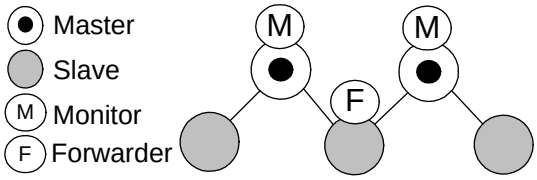

Figure 6.6: Monitor in a slave

If it is not possible to implement a way to forward the topology information, e.g. the slave is a dumb headset; the scatternet will be seen as two separate piconets from the monitors.

How are the monitors aware of other monitors?

In figure 6.5 and 6.6 we saw that we had direct or indirect links between the monitors. But how can the monitor know that it should forward information on a link? In figure 6.5 the device that is master-only does not know that one of its slaves is also a master in another piconet. The master that is also a slave will however know this and it must in some way notify the other master of this.

In 6.6 we have a similar scenario. Neither of the masters will know that it is part of a scatternet and that an inter-piconet link goes through the slave. In this scenario it is the slave that has to notify both masters that it is connected to two masters and that it can act as a forwarder.

Exchange of information

We have now discussed where we need to locate the monitors to get a complete view of the scatternet topology. When this is achieved, we need a way for the monitors to distribute the information of their own piconet, but we must also identify the information needed in the exchanged packets.

The distribution of information will depend on our basic piconet monitor from the previous section. We will take information provided from that and make sure all other monitors get that information. In the piconet monitor each master had knowledge of which slaves that was attached to it. To distribute this information we send this information to every monitor this monitor has a link to. These destination monitors would then record this information and forward it on their own monitor links, except the link where it got the information. In case of cycles in the scatternet we also need a time-stamp, so the same information not will be forwarded from a given node more than once. The exchanged information will include the information provided by the piconet monitor (table 6.1) with an additional time-stamp. This information is shown in table 6.2.

Description	**Type**
Master	BD_ADDR
Slaves	BD_ADDR 1, ..., BD_ADDR N
Time-stamp	Integer

Table 6.2: Scatternet packet information

A device should generate such packets whenever the piconet changes, and send the packet to all the monitors it is connected to (directly or through forwarders). When a device receives a packet, it should compare the time-stamp with the current value. A received packet with new information should be saved and sent to all monitors. If a received packet includes an already saved time-stamp we have a cycle, and the packet should be discarded.

Detection of change

Topology changes in the scatternet should as fast as possible propagate to all monitors. In section 4.1.4 we identified various changes that could occur

within a piconet. If we detect one of the changes from section 4.1.4, it should fire the distribution of the messages explained in the previous section.

In addition to the changes within a piconet that are explained in section 4.1.4 a scatternet topology may change like this:

- A new device creates a piconet
- A piconet cease to exist (powered down or goes out of range)

These changes describe addition and removal of piconets in a scatternet. Regarding changes that can occur within a piconet and with piconets at a higher level we have to discuss:

- Who will detect the topology change?
- How should we check for changes?

We start with the changes within a piconet. The piconet monitor should detect these changes and the distribution of information should start when changes have occurred. How fast the distribution happen depends on the programming model used (i.e. push or pull). With the piconet monitor we had the same issue, but there the push/pull was done on the Bluetooth stack. In the scatternet monitor the push/pull is done on the piconet monitor. If the scatternet monitor is implemented as an extension to the piconet monitor, the changes can probably be caught within the same events as the piconet monitor (i.e. we do not need an additional push or pull). But if we implement the scatternet monitor as an external application it must either check for changes at reasonable intervals (pull) or be notified by events (push). Whether it is done by push or pull the monitor should as fast as possible send the updated Bluetooth network topology information out on the monitor links.

If a piconet goes out of range from the rest of the scatternet, the link between the monitors will be lost. But if nothing is done in the remaining scatternet, the other monitors will not be notified of the removal of this piconet. To solve this every monitor must send out a special *alive* packet that says nothing is changed but the piconet is still connected. If no *alive* packet is received within a given time, we assume that the piconet is no longer connected to the scatternet. *Alive* packets sent too often will result in unnecessary network traffic, but if it is sent too infrequently we will more often get dirty data in the monitors.

If a master is intentionally powered down, we might be able to notify the other monitors that this piconet will no longer be available, before the power down is carried out. But if no such notification is possible the power down will later be noticed by the lack of *alive* packets from this piconet.

When a new master creates a piconet by connecting to one or more devices, a monitor should start up. The monitor should try to connect to other monitors, so information from this piconet could be sent throughout the scatternet.

The master/slave switch could be more difficult to handle, but we propose one simple solution. In a switch the former master stops to monitor and optionally notifies the other monitors about this. The former slave will start to monitor like any other new device that makes an initial connection, as described above.

Implementation of the information push

We have shown that when changes occur, the monitor should as fast as possible push the updated information to all monitors and forwarders it is connected to. In this section we discuss what parts of the Bluetooth stack the information push could utilize. This will only serve as an overview of different solutions to the problems and further studies should be carried out to identify the most appropriate solution.

In the introduction to this chapter we stated that the monitor does not depend on any profiles in the Bluetooth stack, except the General Access Profile. With the scatternet monitor this is not true. The exchange of monitor information can not be carried out without a profile that supports data transportation. The General Access Profile only supports connection management, not exchange of data between devices.

We have identified two possible solutions, both discussed here:

- **New Profile** The advantage of designing a new profile to handle the exchange of information is that we are free to adapt the profile to our exact need. There is however a strong disadvantage that we have to implement the profile on all stacks used throughout the topology.
- **OBEX Push** OBEX Push seems to support what we need to carry out the information push. With the OBEX Push profile *Push clients* can push and pull objects from a *Push server* and this is exactly what we need. There are however one obstacle, OBEX Push are primarily targeted at calendar and business card exchange.

The design of new profiles is complex task and we have chosen not to investigate this solution further in this thesis. Information exchange using OBEX Push, however, seems to be a more straightforward solution and we have therefore taken a closer look at this solution.

As noted, the OBEX Push Profile [15] is at this time primarily targeted at calendar and business card exchange. There are however room for extending

the supported objects. In the OBEX Push Profile we find the OBEX Push Service Record that defines the values returned from a SDP request. The record includes a Supported Formats List with six predefined values (shown in table 6.3).

Key	**Value**
0x01	vCard 2.1
0x02	vCard 3.0
0x03	vCal 1.0
0x04	iCal 2.0
0x05	vNote
0x06	vMessage
0xFF	any type of object

Table 6.3: Supported Formats List

The Supported Formats List can hold 255 different values, and only 7 out of these are used. We propose to use one of the values that are not yet assigned (0x07-0xFE) and use that value to represent our Bluetooth network topology information packets (table 6.2).

In this section we had a short discussion of some implementations the information exchange could use. Further studies should be carried out to conclude what the optimal solution is. The OBEX Push solution should be implemented to see how it performs on common Bluetooth stacks.

Interpretation and presentation

If the exchange of information is successful, every monitor will have enough information to present the topology. The monitor should at least support presentation of raw data that could be used by higher layer- and optionally remote applications. This raw data would be a simple list of all masters and the corresponding slaves in all piconets throughout the scatternet, like the tables in figure 6.7.

It might also desirable to present the information graphically, like shown in figure 6.7.

In this figure interpretation of raw data into a schematically figure representing the Bluetooth network topology is shown. This would be feasible when the network topology information is presented to a person.

Limitations of our approach

We have now finished designing the scatternet monitor and repeat the problem statement to see how well we have answered it:

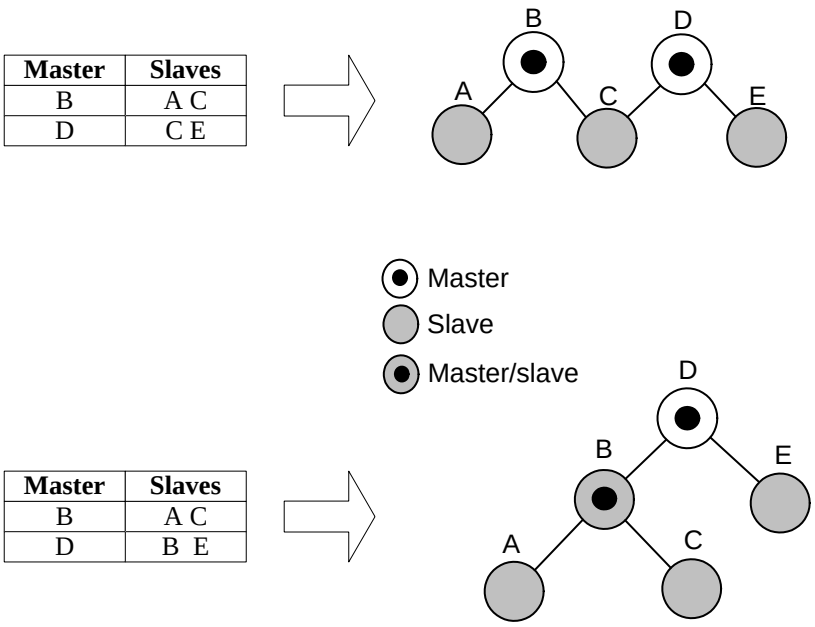

Figure 6.7: Two examples of scatternet presentation

Is it possible to construct a Bluetooth network topology monitor?

An answer to that question could be a simple "Yes". We have however only covered a subset of the topic, the piconet and scatternet. The term "Bluetooth network topology" are however not necessarily limited to devices with connections between them. In figure 6.8 we see two piconets that are *not* connected together.

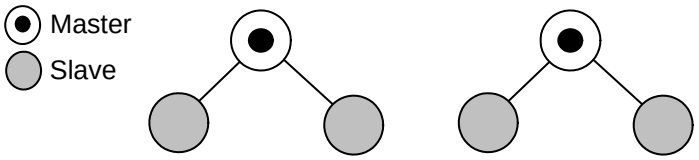

Figure 6.8: Two independent piconets

It can be argued that these piconets are members of the same Bluetooth network topology, they are however not members of the same scatternet. If we monitor it using our scatternet monitor, the two piconets would thus not be aware of each other.

Our piconet and scatternet monitors are therefore not the final answer to the problem statement. They may be used in cases where it is expected that all devices we want monitor are connected together. If we however,

want devices to also be aware of other piconets and scatternets within the geographic area, the scatternet monitor can not be used.

6.4 Bluetooth solution - conclusion

In this chapter we have shown that a Bluetooth Network topology monitor can be constructed using a standard Bluetooth device and Bluetooth stack. We implemented the first part of such a monitor, the Piconet Monitor. The Piconet Monitor is able to monitor all connection changes that occur within a single piconet.

We also designed the scatternet monitor that used information collected by the piconet monitor to monitor a scatternet. The scatternet monitor was however not exactly what we wanted. It can not monitor piconets that are not connected together as a scatternet. In these cases we have not found any solution to the Bluetooth network monitoring.

Chapter 7

Conclusion

In this last chapter we summarize the thesis with achievements and results. The chapter also includes a critical review of our work and some pointers to further research on the topic.

7.1 Achievements and results

The work on this thesis was conceived while studying the details of the Bluetooth technology. We found that although the Bluetooth specification included the concept of scatternets, there was no way for applications to obtain information about the Bluetooth network topology. Without information about the network topology the scatternet will only be a term, not a tool that application developers could use to construct advanced Bluetooth applications.

With detailed knowledge of the Bluetooth specification we investigated two ways to answer the problem statement; "*Is it possible to construct a Bluetooth network topology monitor?*". In both approaches we stated that it was feasible to design the monitor as a two-layered application, with a piconet monitor at the bottom and a scatternet monitor collecting information from piconet monitors.

7.1.1 Hardware approach

We first looked at the approach with a custom-built hardware device. The intention was that the custom-built device could eavesdrop on the radio band that Bluetooth operates and deduce the topology from the captured traffic. To decide whether this was possible we had to study thoroughly how the Bluetooth technology utilizes the radio frequencies.

To carry out the eavesdropping of Bluetooth traffic the custom-built device first had to *synchronize to the piconet* and then *capture the raw traffic*. We

took a closer look at protocol analyzers to discuss synchronization and the Bluetooth specification to discuss the analysis of captured traffic. From this study we found that neither the synchronization nor the analysis of the traffic is trivial to implement:

1. The synchronization against piconets is hard to accomplish without prior knowledge of the Bluetooth network topology.
2. Normal communication does not reveal the 48-bit MAC address (BD_ADDR) that is needed to identify Bluetooth devices. The BD_ADDR is only revealed in the special FHS packets, and because the synchronization is not trivial, the capture of all FHS packets will be very hard.

Because of these findings we did not continue the design of the Bluetooth network topology monitor that was based on custom-built hardware.

7.1.2 Software approach

In this approach we looked into using standard Bluetooth devices and the services provided by a Bluetooth stack.

Piconet monitor

We discovered that information about local connections could be extracted from the Bluetooth stack. A piconet master may then easily obtain the piconet network topology, and we thus placed the piconet monitor in the master of the piconet (figure 7.1).

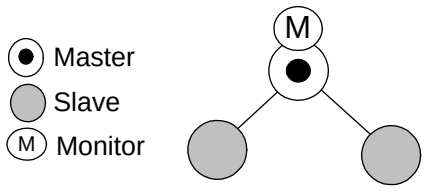

Figure 7.1: Monitoring one piconet

An implementation of the piconet monitor was carried out, and that gave us valuable feedback to the design phase.

Scatternet monitor

In addition to the piconet monitor we implemented, we designed the scatternet monitor as a distributed application. We assumed that piconet monitors was present in every piconet, monitoring their own piconet. The theory is that every piconet monitor should exchange piconet information with each

other. If this exchange succeeds, every piconet monitor would obtain information about the whole scatternet.

In some scatternets (figure 7.2) there are direct links between monitors, and these links can be used to exchange the information.

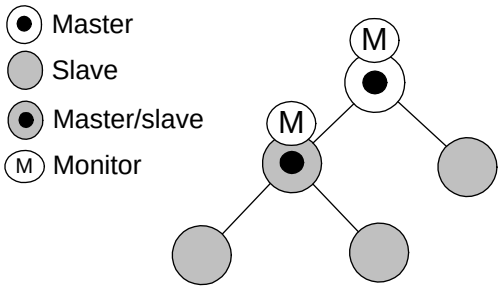

Figure 7.2: Scatternet monitor 1

There are however, cases where there is no such direct connection (figure 7.3). This happens when a slave is a member of more than one piconet, and to solve these scenarios we introduced the *forwarder*.

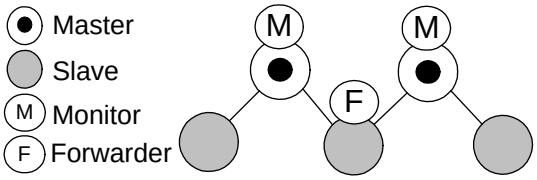

Figure 7.3: Scatternet monitor 2

The job of the forwarder is to forward the received information to the other masters it is connected to. All monitors are now either directly or indirectly connected, and the exchange of information can be carried out.

We also gave some indications on how the exchange of information should be carried out. The OBEX Push profile was suggested, but further studies should be carried out to verify if this is the most optimal solution.

7.1.3 Did we answer the problem statement?

As a final note in chapter 6, we noted that our solution did not fully answer the problem statement. The reasion is that two piconets can be present in the same space without being members of the same scatternet (figure 7.4).

We have however answered an important subset of the problem statement, and leave it as further research to look more closely at topologies where the topology is not a scatternet.

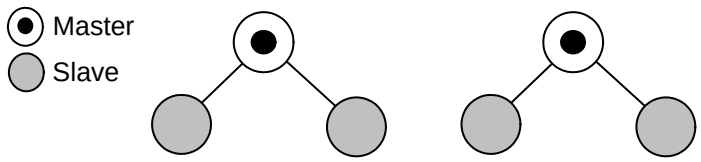

Figure 7.4: Two independent piconets

7.2 Critical review of our work

As noted, the work on the thesis was started to gain better understanding of the Bluetooth network topology and ended up specially monitoring scatternet topology. While writing the thesis we have however, become more skeptical to the whole scatternet concept. Although specified, the scatternet has not been widely implemented in Bluetooth stacks. Devices might also initiate connections when they need them and tear them down after use. This may indicate that Bluetooth do not need the advanced network topologies that the scatternet provides. The piconet may be enough, because exchange of data between Bluetooth devices will not go via other Bluetooth devices. It will either go directly between the devices or through other backbone networks. Time will show whether this will be the case.

In addition we should have used more time discussing the Bluetooth network topology. However, this would not have given us the time to go deeply into the piconet and scatternet monitor.

7.3 Further Research

During the work on this thesis we have identified several interesting and unresolved issues. These issues are out of the scope of this thesis, but would be interesting to look into as a continuation of our work:

- We have successfully designed the piconet and scatternet monitor, but our answer does not fully answer the problem statement. Further studies should be carried out to investigate Bluetooth network topology monitoring on topologies that are not connected together as scatternets.

- In chapter 6 we discussed how the information push between monitors in a scatternet monitor scenario could be carried out. Our conclusion was that the OBEX Push profile could be used. This issue should, however, be investigated further and preferably an implementation of our proposed solution should be done. These investigations could however, be a complex task, and we did not have time to include this in the thesis.

- The software monitor we designed included a data structure that holds information about the network topology. This data structure is however not available outside the application. It would be feasible to add an API to our monitor that could provide information about the topology to external applications.

- As part of the software monitor, we implemented the first step, the piconet monitor. It would also be interesting to implement the scatternet monitor, but that was not possible with the development environment we used (Digianswer). Further investigations on other Bluetooth stacks could also be carried out, to get a status of scatternet support in other Bluetooth stacks.

- We have shortly presented the motivation behind our Bluetooth network topology monitor. It may i.e. be used to construct routing tables that can be used in Bluetooth scatternet routing. The different use-cases could be discussed in more detail to possibly better understand the requirements for different use-cases and how they may benefit from a Bluetooth network topology monitor.

Appendix A

Glossary

AES Advanced Encryption Standard

AM_ADDR Active Member Address
3 bit unique Bluetooth address within a piconet.

API Application Programming Interface

BD_ADDR Bluetooth Device Address
48 bit address unique among all Bluetooth units.

COM Component Object Model
Object Model used on the Microsoft platform.

CRC Cyclic Redundancy Check
Method of checking for errors in data that has been transmitted on a communications link [33].

DAC Device Access Code
A code which identifies a particular Bluetooth device. This code is derived from the device's Bluetooth device address and is used when paging the device. [6]

DECT Digital Enhanced Cordless Telecommunications
Digital wireless telephone technology.

DES Data Encryption Standard

DSSS Direct Sequence Spread Spectrum

FHS Frequency Hopping Sequence
Bluetooth packet that carries synchronization data.

FHSS Frequency Hopping Spread Spectrum
Modulation technique which spreads data across the entire transmission spectrum by transmitting successive data on different channels, or "hopping". [6]

GAP Generic Access Profile
Basic set of functions shared by all Bluetooth devices.

GIAC General Inquiry Access Code
A fixed standard code used to inquire for devices; its value is 0x9E8B33. [6]

GSM Global System for Mobile communications
2nd generation cellular phone system.

HCI Host Controller Interface
Logical partition between Bluetooth host and Bluetooth module.

IAC Inquiry Access Code
Sent in ID packets by devices wanting to discover other Bluetooth devices in the area. [6]

ISM Industrial Scientific Medical
Common frequency range reserved for industrial and scientific use.

L2CAP Logical Link Control and Adaption
Layer in the Bluetooth protocol stack which deals with multiplexing, segmentation and reassembling.

LAN Local Area Network

LAP Lower Address Part
Lower part of the BD_ADDR.

LLC Logical Link Control

MTU Max Transfer Unit
The largest payload a particular layer in a protocol stack can handle.

NAP Non-significant Address Part
Non-significant part of the BD_ADDR.

OFDM Orthogonal Frequency Division Multiplexing
A modulation technique that transmits data across many carriers for high data rates at lower symbol rates. [6]

PAN Personal Area Networking
Small ad-hoc network that join personal devices in a network.

QoS Quality of Service
Reservation of various quality metrics on a communication link.

RSSI Received Signal Strength Indication
Measuring of the received signal strength.

SDP Service Discovery Profile
Bluetooth profile that allows a client to locate services in a Bluetooth device.

SIG Bluetooth Special Interest Group
The group of companies who have registered an interest in Bluetooth via the SIG website. [6]

SWAP Shared Wireless Access Protocol
Layer in the HomeRF stack which supports voice and data traffic.

UAP Upper Address Part
Upper part of the BD_ADDR.

Appendix B

Piconet Monitor - Code listing

Listing B.1: Monitor.h

```
/*

    Bluetooth piconet network topology monitor

    FILE: Monitor.h
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    Main header file for the Monitor application
    This file is the header file to Monitor.cpp.

*/

#if !defined(MONITOR_H)
#define MONITOR_H

#pragma once

#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h" // main symbols

// See Monitor.cpp for the implementation of this class
class CMonitorApp : public CWinApp
{
```

```
public:
    CMonitorApp();

// Overrides
public:
    virtual BOOL InitInstance();

};

#endif // !defined(MONITOR_H)
```

Listing B.2: Monitor.cpp

```
/*

    Bluetooth piconet network topology monitor

    FILE: Monitor.cpp
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    Monitor.cpp instatites the Bluetooth COM pointer and
    starts the main dialog (MonitorDlg.cpp)

*/

#include "stdafx.h"
#include "Monitor.h"
#include "MonitorDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// CMonitorApp constructor (empty)
CMonitorApp::CMonitorApp()
{
}

CMonitorApp theApp;

// The BT COM object
CComModule _Module;

// CMonitorApp initialization
BOOL CMonitorApp::InitInstance()
{
    CWinApp::InitInstance();
    OleInitialize (NULL);
    {
        // Due to this scope, the m_pBluetooth smart pointer
        // in CMonitorDlg delete itself and release the
        // IBluetoothEvents interface
        // Without this scope the CoUninitialize will be
        // called before all com object actualy were released.
```

```
        _Module.Init(NULL, m_hInstance); // Init ATL com module
        CMonitorDlg dlg;                 // New main dialog
        m_pMainWnd = &dlg;
        int nResponse = dlg.DoModal(); // Show the main dialog
        _Module.Term();
    }
    OleUninitialize();

    // Since the dialog has been closed, return FALSE so that
    // we exit the application, rather than start the
    // application's message pump.

    return FALSE;
}
```

Listing B.3: MonitorDlg.h

```
/*

    Bluetooth piconet network topology monitor

    FILE: MonitorDlg.h
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    This file is the header file to MonitorDlg.cpp.
    Here are various functions declared in addittion to
    our BTPiconet struct.

*/

#if !defined(MONITORDLG_H)
#define MONITORDLG_H

#pragma once

#include "eventhandler.h"
#include "stdafx.h"
#include "Monitor.h"
#include "BTAddress.h"

// This is the struct we use to store information
// about the piconet that _we_ are master in

struct BTPiconet
{
    // Constructor
    BTPiconet() {
        slaveCount = 0;
        master = NULL;
    }
    char * master;   // master BD_ADDR
    char * slave [7]; // BD_ADDR of the slaves
    int slaveCount;  // number of slaves in this piconet
};

class CEventHandler;

// About dialog box
class CAboutDlg : public CDialog
{
```

```
public:
    CAboutDlg();

    enum { IDD = IDD_ABOUTBOX };

protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV
};

// CMonitorDlg dialog (Main application)
class CMonitorDlg : public CDialog
{
public:
    CMonitorDlg(CWnd* pParent = NULL); // Standard constructor

    // The MonitorUpdate() function is called from
    // eventhandler.cpp, so it have to be public
    void MonitorUpdate(void);

protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV
    HICON m_hIcon;

    // Generated message map functions
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    DECLARE_MESSAGE_MAP()

private:
    CListBox output; // The output listbox
    void BluetoothInit(void);
    void MonitorOutput(BTPiconet piconet);
    void SetCaption(const char * arg);
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnTimer(UINT nIDEvent);

    // Dialog Data
    enum { IDD = IDD_MONITOR_DIALOG };
    CComObject<CEventHandler> * m_pEvHandler; // pointer to event sink
    dgaCOM::IBluetoothPtr m_pBluetooth;
};

#endif // !defined(MONITORDLG_H)
```

Listing B.4: MonitorDlg.cpp

```
/*

    Bluetooth piconet network topology monitor

    FILE: MonitorDlg.cpp
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    This file is the main file, where the Bluetooth module
    is initialized and the connections extracted.

*/

#include "MonitorDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// About box constructor (do nothing)
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}

// Data exchange for about box
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

// CMonitorDlg (main) dialog
CMonitorDlg::CMonitorDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CMonitorDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

// Data exchange for main dialog
void CMonitorDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    // Map the listbox (output) to the IDC_OUTPUT resource
```

```
    DDX_Control(pDX, IDC_OUTPUT, output); // Our listbox
}

BEGIN_MESSAGE_MAP(CMonitorDlg, CDialog)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()

    // Timer maps
    ON_WM_CREATE()
    ON_WM_TIMER()
END_MESSAGE_MAP()


// This function is called when the main dialog
// is shown.
// It sets up the environment and call the function
// that initializes the Bluetooth device
BOOL CMonitorDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
                strAboutMenu);
        }
    }

    // Set the icon for this dialog.
    SetIcon(m_hIcon, TRUE); // Set big icon
    SetIcon(m_hIcon, FALSE); // Set small icon

    // Initialize Bluetooth device
    BluetoothInit();

    return TRUE;
}
```

```
// This functions is called when the user selects a
// command from the title-line (e.g. about box)
void CMonitorDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    // Our about box
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}

// Init Bluetooth device
void CMonitorDlg::BluetoothInit(void)
{
    try
    {
        HRESULT hr = m_pBluetooth.CreateInstance(dgaCOM::CLSID_Bluetooth);

        if (SUCCEEDED(hr))
        {
            m_pEvHandler = 0;

            // Create the event sink object for event fired by IBluetooth
            CComObject<CEventHandler>::CreateInstance(&m_pEvHandler);
            m_pEvHandler->SetDialogPtr(this);

            m_pEvHandler->Connect(m_pBluetooth);
        }
        else
        {
            output.AddString("Failed to crate Bluetooth pointer!");
        }
    }
    catch (_com_error ce)
    { // pcmcia card not inserted?
        output.ResetContent();
        output.AddString("COM Error!");
        output.AddString("Ensure that the software is correctly installed and");
        output.AddString("the Bluetooth device plugged in!");
    }
}

// Function to set title of main window
void CMonitorDlg::SetCaption(const char * arg)
```

```
{
    char s[255];
    if (arg)
    {
        if (strlen(arg) > 0)
            sprintf (s, "%s (%s)", DLG_CAPTION, arg);
        else
            sprintf(s, "%s", DLG_CAPTION);
        SetWindowText(s);
    }
}

// Get active connections from the COM object and store
// the results in a BTPiconet struct
void CMonitorDlg::MonitorUpdate(void)
{
    HRESULT hr;
    dgaCOM::BTConnectionRecord *connections = NULL;
    unsigned short connectionCount, i;
    BD_ADDR local_bdaddr;
    CBTAddress * local_addr;
    CBTAddress * remote_addr;
    BTPiconet piconet;

    try
    {
        // Get local address.
        // We do this on every update, to make sure that the
        // device address is correct if the Bluetooth device
        // is inserted after the monitor has started

        hr = m_pBluetooth->GetLocalDeviceAddress(&local_bdaddr);
        local_addr = new CBTAddress(local_bdaddr,0);
        piconet.master = (char *) local_addr->GetAddrAsString();
        free(local_addr);

        // Get list of active conncections
        hr = m_pBluetooth->GetConnections(&connections, &connectionCount);
        if (SUCCEEDED(hr))
        {
            for (i = 0; i < connectionCount; i++)
            {
                remote_addr = new CBTAddress(connections[i].Address,0);

                // Only interested in links where we are master
                if (connections[i].Master == 0)
                {
                    piconet.slave[piconet.slaveCount] = (char *) remote_addr->
                        GetAddrAsString();
```

```
                    piconet.slaveCount++;
                }
                free(remote_addr);
            }

            // Free mem allocated by COM
            CoTaskMemFree(connections);
        }
    }
    catch (_com_error ce)
    { // pcmcia card not inserted?
        output.ResetContent();
        output.AddString("COM Error!");
        output.AddString("Ensure that the software is correctly  installed  and");
        output.AddString("the Bluetooth device plugged in!");
    }

    // Output to screen
    MonitorOutput(piconet);
}

// Output the piconet monitor information to the screen
void CMonitorDlg::MonitorOutput(BTPiconet piconet)
{
    CString str;
    int i;

    Beep(200,200); // Beep on every update

    if (piconet.master == NULL)
    {
        SetWindowText("Failed initializing Bluetooth device!");
    }
    else
    {
        str.Format(_T("Piconet Monitor for %s"), piconet.master);
        SetWindowText(str);
    }

    output.ResetContent(); // Empty the output window

    for (i = 0; i < piconet.slaveCount; i++)
    {
        str.Format(_T("%d -> %s"), i, piconet.slave[i]);
        output.AddString(str);
    }

    if (piconet.slaveCount == 0)
        output.AddString("<NONE>");
```

```
}

// Initialize a timer
// The timer will start the function CMonitorDlg::OnTimer()
int CMonitorDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CDialog::OnCreate(lpCreateStruct) == -1)
        return -1;

    SetTimer(1,10000,NULL); // Fire timer every 10000ms (10s)

    return 0;
}

// Function that is ran every 10000ms when initialized by
// CMonitorDlg::OnCreate()
void CMonitorDlg::OnTimer(UINT nIDEvent)
{
    MonitorUpdate(); // Manually check for changes in the connection
    CDialog::OnTimer(nIDEvent);
}
```

Listing B.5: eventhandler.h

```
/*

    Bluetooth piconet network topology monitor

    FILE: eventhandler.h
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    This file is the header file to eventhandler.cpp.

*/

#if !defined(EVENTHANDLER_H)
#define EVENTHANDLER_H

#include "stdafx.h"
#include "MonitorDlg.h"

class CMonitorDlg;

class ATL_NO_VTABLE CEventHandler :
    public CComObjectRoot,
    public dgaCOM::IBluetoothEvents
{
public:
    BEGIN_COM_MAP(CEventHandler)
        COM_INTERFACE_ENTRY(dgaCOM::IBluetoothEvents)
    END_COM_MAP()

    CEventHandler();
    virtual ~CEventHandler();

    HRESULT Connect(IUnknown * pUnk);
    HRESULT Disconnect(IUnknown * pUnk);
    void SetDialogPtr(CMonitorDlg * dialog);

    STDMETHOD(raw_OnInquiryComplete) ( );
    STDMETHOD(raw_OnInquiryResult) (dgaCOM::BTInquiryResult InquiryResult);
    STDMETHOD(raw_OnInquiryStart) ( );
    STDMETHOD(raw_OnInquiryCancel) ( );
    STDMETHOD(raw_OnConnection) (dgaCOM::BTConnectionRecord ConnectionInfo)
        ;
    STDMETHOD(raw_OnDisconnection) (dgaCOM::BTConnectionRecord
        ConnectionInfo);
```

```
    STDMETHOD(raw_OnLocalDeviceRemoved) ( );
    STDMETHOD(raw_OnLocalDeviceReset) ( );
    STDMETHOD(raw_OnRemoteNameResult) (BD_ADDR Address, dgaCOM::
        BT_Name FriendlyName);
    STDMETHOD(raw_OnDeviceListChange) (BD_ADDR Address);
    STDMETHOD(raw_OnCloseDown) ( );

private:
    CMonitorDlg * dialog;
    DWORD m_dwCookie;
};

#endif // !defined(EVENTHANDLER_H)
```

Listing B.6: eventhandler.cpp

```
/*

    Bluetooth piconet network topology monitor

    FILE: EventHandler.cpp
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    This file is the implementation of the events that
    is fired by the BT COM object.

    We use only
    CEventHandler::raw_OnConnection()
    ,
    CEventHandler::raw_OnDisconnection()
    og
    CEventHandler::raw_OnLocalDeviceRemoved()

*/

#include "stdafx.h"
#include "EventHandler.h"

CEventHandler::CEventHandler()
{
}

CEventHandler::~CEventHandler()
{
}

void CEventHandler::SetDialogPtr(CMonitorDlg * dialogPtr)
{
    dialog = dialogPtr;
}

// On first connection
HRESULT CEventHandler::Connect(IUnknown * pUnk)
{
    dialog->MonitorUpdate();
    return AtlAdvise(pUnk, this->GetUnknown(), dgaCOM::IID_IBluetoothEvents, &
        m_dwCookie);
}
```

```
HRESULT CEventHandler::Disconnect(IUnknown * pUnk)
{
    return AtlUnadvise(pUnk, dgaCOM::IID_IBluetoothEvents, m_dwCookie);
}

// On connection to other Bluetooth devices
STDMETHODIMP CEventHandler::raw_OnConnection (dgaCOM::BTConnectionRecord
    ConnectionInfo)
{
    TRACE("CEventHandler::raw_OnConnection\n");
    dialog->MonitorUpdate();
    return S_OK;
}

// On disconnection to other Bluetooth devices
STDMETHODIMP CEventHandler::raw_OnDisconnection (dgaCOM::
    BTConnectionRecord ConnectionInfo)
{
    TRACE("CEventHandler::raw_OnDisconnection\n");
    dialog->MonitorUpdate();
    return S_OK;
}

STDMETHODIMP CEventHandler::raw_OnInquiryComplete( )
{
    TRACE("CEventHandler::raw_OnInquiryComplete\n");
    return S_OK;
}

STDMETHODIMP CEventHandler::raw_OnInquiryResult (dgaCOM::BTInquiryResult
    InquiryResult)
{
    TRACE("Device found.....Address");
    return S_OK;
}

STDMETHODIMP CEventHandler::raw_OnInquiryStart ( )
{
    TRACE("CEventHandler::raw_OnInquiryStart\n");
    return S_OK;
}

STDMETHODIMP CEventHandler::raw_OnInquiryCancel ( )
{
    TRACE("CEventHandler::raw_OnInquiryCancel\n");
    return S_OK;
}

STDMETHODIMP CEventHandler::raw_OnLocalDeviceRemoved ( )
```

```
{
    TRACE("CEventHandler::raw_OnLocalDeviceRemoved\n");
    dialog->MonitorUpdate();
    return S_OK;
}

STDMETHODIMP CEventHandler::raw_OnLocalDeviceReset ( )
{
    TRACE("CEventHandler::raw_OnLocalDeviceResed\n");
    return S_OK;
}

STDMETHODIMP CEventHandler::raw_OnCloseDown ( )
{
    TRACE("CEventHandler::raw_OnCloseDown\n");
    return S_OK;
}


STDMETHODIMP CEventHandler::raw_OnRemoteNameResult (BD_ADDR Address,
     dgaCOM::BT_Name FriendlyName)
{
    TRACE("Done name lookup...");
    return S_OK;
}

STDMETHODIMP CEventHandler::raw_OnDeviceListChange (BD_ADDR Address)
{
    TRACE("CEventHandler::raw_OnDeviceListChange\n");
    dialog->MonitorUpdate();
    return S_OK;
}
```

Listing B.7: BTAdress.h

```
/*

    Bluetooth piconet network topology monitor

    FILE: BTAddress.h
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    This file is the header file to BTAddress.cpp.

*/

#if !defined(BTADDRESS_H)
#define BTADDRESS_H

#pragma once

#include "stdafx.h"
#include <string>
#include <windows.h>

class CBTAddress {
public:
    // Address as "00:50:CD:10:00:5D" or 0050CD10005D
    CBTAddress(const char * szAddr);
    // Address as pointer to array of six unsigned chars
    CBTAddress(BD_ADDR addr, unsigned long ClassOfDevice);
    // Returns address as "00:50:CD:10:00:5D"
    const char * GetAddrAsString();
    // Returns the address
    void GetAddr(BD_ADDR addr);
    // Sets the FriendlyName
    void SetFriendlyName(const char *newFriendlyName);
    // Returns the FriendlyName
    const char * GetFriendlyName();
    // Returns Class of Device/Service
    unsigned long GetClassOfDevice(void);

    CBTAddress(BD_ADDR addr);


private:
    BD_ADDR m_TheAddress;
    unsigned long m_TheClassOfDevice;
```

```
    std::string m_TempStr;
    std::string m_FriendlyName;

    unsigned char HexToInt(char ch);
};

#endif !defined(BTADDRESS_H)
```

Listing B.8: BTAdress.cpp

```
/*

    Bluetooth piconet network topology monitor

    FILE: BTAddress.cpp
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    This class is a simple wrapper for a Bluetooth address.
    It holds some nice-to-have-functions (eg. GetAddrAsString)

*/


#include "BTAddress.h"

// Address as "5D:00:10:CD:50:00"
CBTAddress::CBTAddress(const char * szAddr)
{
    m_TheAddress[0] = HexToInt(szAddr[15])*16+HexToInt(szAddr[16]);
    m_TheAddress[1] = HexToInt(szAddr[12])*16+HexToInt(szAddr[13]);
    m_TheAddress[2] = HexToInt(szAddr[9])*16+HexToInt(szAddr[10]);
    m_TheAddress[3] = HexToInt(szAddr[6])*16+HexToInt(szAddr[7]);
    m_TheAddress[4] = HexToInt(szAddr[3])*16+HexToInt(szAddr[4]);
    m_TheAddress[5] = HexToInt(szAddr[0])*16+HexToInt(szAddr[1]);
}

// Address as pointer to array of six unsigned chars
CBTAddress::CBTAddress(BD_ADDR Addr, unsigned long ClassOfDevice)
{
    for (int i = 0; i < 6; i++)
        m_TheAddress[i] = Addr[i];

    unsigned char CoD[3];
    CoD[0] = (unsigned char) ClassOfDevice » 16;
    CoD[1] = (unsigned char) ClassOfDevice » 8;
    CoD[2] = (unsigned char) ClassOfDevice;

    m_TheClassOfDevice = CoD[0] + (CoD[1]«8) + (CoD[2]«16);
}

// Returns address as "00:50:CD:10:00:5D"
const char * CBTAddress::GetAddrAsString()
{
```

```
    char buf[25];
    sprintf(buf, "%02X:%02X:%02X:%02X:%02X:%02X", m_TheAddress[5],
        m_TheAddress[4], m_TheAddress[3], m_TheAddress[2], m_TheAddress[1],
        m_TheAddress[0]);
    m_TempStr = buf;
    return m_TempStr.c_str();
}

void CBTAddress::GetAddr(BD_ADDR addr)
{
    for (int i = 0; i< 6; i++)
        addr[i] = m_TheAddress[i];
}

// Sets the FriendlyName
void CBTAddress::SetFriendlyName(const char *newFriendlyName)
{
    m_FriendlyName = newFriendlyName;
}

// Returns the FriendlyName
const char * CBTAddress::GetFriendlyName()
{
    return m_FriendlyName.c_str();
}

unsigned char CBTAddress::HexToInt(char ch)
{
    if (ch >= '0' && ch <= '9')
        return ch-48;
    else
        return ch-55;
}

unsigned long CBTAddress::GetClassOfDevice(void)
{
    return m_TheClassOfDevice;
}

CBTAddress::CBTAddress(BD_ADDR Addr)
{
    for (int i = 0; i < 6; i++)
        m_TheAddress[i] = Addr[i];
}
```

Listing B.9: stdafx.h

```
/*

    Bluetooth piconet network topology monitor

    FILE: stdafx.h
    AUTHOR: Fredrik Borg <fredrikb@@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    Include file for standard system include files,
    or project specific include files that are used frequently,
    but are changed infrequently

*/

#if !defined(STDAFX_H)
#define STDAFX_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define VC_EXTRALEAN  // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>  // MFC core and standard components
#include <afxext.h>  // MFC extensions
#include <afxdtctl.h> // MFC support for IE 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h> // MFC support for Windows Common Controls

const static unsigned char INQUIRY_LENGTH = 10; // search for 10 secs
const static char * DLG_CAPTION       = "Bluetooth API Demo";

#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>

#include "..\..\tl_headers\dgabtapi.tlh"
#include "..\..\tl_headers\dgaBluetoothComObject.tlh"
#include "..\..\tl_headers\dgaSerialPortProfile.tlh"

#endif // _AFX_NO_AFXCMN_SUPPORT

#pragma warning (disable : 4786) // Disable unusable STL warning
```

```
//{{AFX_INSERT_LOCATION}}

#endif // !defined(STDAFX_H)
```

Listing B.10: stdafx.cpp

```
/*

    Bluetooth piconet network topology monitor

    FILE: stdafx.cpp
    AUTHOR: Fredrik Borg <fredrikb@ifi.uio.no>

    Parts of the source code is taken from the Digianswer
    Bluetooth Software Suite (http://www.btsws.com)

    The rest is written as part of my Cand.scient thesis.

    Source file that includes just the standard includes.

*/

#include "stdafx.h"
```

Bibliography

[1] Digianswer.
http://www.digianswer.com.

[2] BPA100 Bluetooth Protocol Analyzer. Tektronix.
http://www.tek.com.

[3] CATC Merlin Bluetooth Protocol Analyzer. Computer access technologies corporation.
http://www.catc.com.

[4] Fredrik Borg, Do van Thanh, and Tore Jønvik. Monitoring bluetooth network topology. *5th World Multi-conference on Systemics, Cybernetics and Informatics (SCI 2001), July 22-25 2001, Orlando, USA*.

[5] Fredrik Borg, Do van Thanh, and Tore Jønvik. Monitoring bluetooth network topology. *6th CDMA International Conference (CIC 2001), October 30. to November 2, Seoul, Korea*.

[6] Jennifer Bray and Charles F. Sturman. *Bluetooth - Connect Without Cables*. Prentice Hall, 2001.

[7] Charles R. Buffler and Per O. Risman. Compatibility issues bewteen bluetooth and high power systems in the ism bands. *Microwave Journal*, july 2000.

[8] CATC. Catc merlin bluetooth protocol analyzer user's manual.
http://www.catc.com.

[9] IrDA Consortium. Irda suite of specifications.
http://www.irda.org.

[10] Digianswer. Bluetooth software suite sdk manual.
http://www.btsws.com.

[11] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4.
http://www.eyetap.org/~rguerra/toronto2001/rc4_ksaproc.pdf [03.09.01].

[12] Electronic Frontier Foundation. Eff des cracker project.
http://www.eff.org/descracker.html [29.08.01].

[13] Jim Geier. *Wireless LANs, Implementing Interoperable Networks.* Macmillan Technical Publishing, 1st edition, 1999.

[14] Bluetooth Special Interest Group. Specification of the bluetooth system - core.
http://www.bluetooth.com.

[15] Bluetooth Special Interest Group. Specification of the bluetooth system - profiles.
http://www.bluetooth.com.

[16] William Gurley. Bye-bye, bluetooth.
http://news.cnet.com/news/0-1270-210-6832075-1.html [20.08.01].

[17] Jaap C. Haartsen and Stefarn Zürbes. Bluetooth voice and data performance in 802.11 ds wlan enviroment.
http://www.palowireless.com.

[18] IEEE. Ieee 802 standards.
http://standards.ieee.org.

[19] Eleven Engineering Inc. Xi spike wireless.
http://www.spike-wireless.com [28.05.01].

[20] Martin Johnsson. Hiperlan/2 - the broadband radio transmission technology operating in the 5 ghz frequency band.
http://www.hiperlan2.com.

[21] RSA Laboratories. Rsa laboratories' frequently asked questions about today's cryptography, version 4.1.
http://www.rsasecurity.com, 2000.

[22] Richard C. Leinecker and Tom Archer. *Visual C++ 6 Bible.* IDG, 1998.

[23] Nathan J. Muller. *Bluetooth Demystified.* McGraw-Hill, 2000.

[24] Palowireless. Wireless resource center.
http://www.palowireless.com.

[25] Joakim Persson and Jaap C. Haartsen. Enabling wireless connections using bluetooth. *Telektronikk 97*, 2001.

[26] Peter Rysavy. Wireless wonders coming your way.
http://www.rysavy.com.

[27] Counterpane Internet Security. The blowfish encryption algorithm.
http://www.counterpane.com.

[28] Ian Sommerivlle. *Software Engineering*. Addison-Wesley, 5th edition, 1995.

[29] Adam Stubblerfield, John Ioannidis, and Aviel D. Rubin. Using the fluhrer, mantin, and shamir attack to break wep.
http://www.cs.rice.edu/~astubble/wep/wep_attack.html [03.09.01].

[30] Arca Technologies. Arca wavecatcher manual.
http://www.arcatech.com.

[31] Tektronix. Synchronizing to bluetooth piconets with the bluetooth protocol analyzer (bpa100).
http://http://www.tek.com.

[32] ARCA Wavecatcher. Arca technologies.
http://www.arcatech.com.

[33] Whatis.com.
http://whatis.techtarget.com.

[34] Jim Zyren. Reliability of ieee 802.11 hi rate dsss wlan in a high density bluetooth environment.
http://grouper.ieee.org/groups/802/15/pub/1999/Sep99/99098r0P802-15_T%G1-Reliability-of-P802-11-DS-BT_JZyren-Intersil.pdf [20.08.01].